MASTER'S THESIS

Model checking protocol implementations: A new adaptation-based approach

Slob, F.

Award date:
2021

Model checking protocol implementations: A new adaptation-based approach

Florian Joost Slob

Student:
Date: 13/03/2021

MODEL CHECKING PROTOCOL IMPLEMENTATIONS: A NEW ADAPTATION-BASED APPROACH

by

Florian Joost Slob

Version:	1.1	
Student number:		
Course code:	IM9906	
Thesis committee:	S. T. Q. Jongmans (chairman and supervisor),	Open University
	Prof. dr. M. C. J. D. van Eekelen (second reader),	Open University

CONTENTS

SUMMARY

Multi-core computer systems can simultaneously run multiple threads, and developers must create concurrent programs and implement protocols for interaction between those threads. This error-prone task requires significant knowledge and skill from developers.

A possible approach to reduce concurrency bugs is moving communication-specific code to isolated protocol modules that can be tested separately from the other logic of the application. To overcome the difficulties of writing those protocol modules manually, researchers have created domain-specific languages (DSL) and other programming and testing tools. A DSL protocol module can be compiled into a general-purpose language (GPL) protocol module. This does not ensure valid protocol code, but it enables developers to easily separate the protocol code from the threads written in GPL code.

A sophisticated and powerful technique that many DSLs for protocol programming support is model checking. With model checking, one can verify user-specified assertions with the certainty of full coverage of the state-space. Current techniques convert the protocol definition (written in that DSL) to an abstract model and use it as input for external model checkers. There are, however, drawbacks to this approach. Firstly, there are dependencies on external third-party tools. External tools can lose support, have breaking changes in updates, and make integration into a modern build and integration pipeline more difficult. Secondly, since this approach uses an abstract model, there is a need to validate the abstraction techniques. If an assertion holds for the abstract model, one would need additional evidence to transfer verification results to the original protocol code.

In this thesis, we develop a different approach. Instead of using abstract models, we use real executions of protocol implementations to construct state-spaces for model checking.

To provide evidence for this concept, we created Prut4j: a tool to simplify exhaustive unit testing of channel/queue-based communication protocols in concurrent Java programs, using model checking. Prut4j consists of two DSLs for writing "protocol modules" and unit tests, a code generator to generate protocol implementations in Java from the DSL syntax, and a runtime environment to execute the unit tests. Developers can write unit tests as LTL formulas expressed in a DSL. The runtime environment uses the protocol implementations (in Java) to construct a state-space for model checking. The state-space is generated through real execution of the protocol implementation code in an on-the-fly manner while checking the LTL formulas' correctness.

We conducted three different case studies to validate Prut4j and show that we can generate complex protocol implementations and test sophisticated user-defined assertions using the model checking techniques.

Moreover, our research shows that Prut4j-based programs perform well in a third-party benchmark. Finally, we show that model checking protocol code can be easier using Prut4j since there is no dependency on external tools.

With the development and analysis of Prut4j, we have given first evidence of a novel approach to model checking protocol implementations and have shown that it is feasible and worth further research and development. A paper based on this thesis is accepted for publication and presentation at ICST 2021.

1

INTRODUCTION

Many software systems cannot simply be shut down or restarted when in an error state in order to roll back to a safe configuration, such as, for example, software in embedded systems like cars and airplanes. We increasingly depend on critical software systems in our daily lives. Ensuring correct behavior of critical software systems is therefore essential. With the rapidly advancing software technology, it is increasingly important to develop methods that enable us to reason about software behavior and increase confidence in software correctness [9, 24].

Many modern software systems consist of threads that communicate with each other, and concurrent programming is essential as multicore technology advances. Concurrent programs on modern computer systems can locally run multiple threads at the same time to increase performance.

As a result of these developments, communication between threads, is found everywhere today, but also complex to get right. Notably, programmers must implement protocols for interaction between threads; this task is error-prone and requires significant knowledge about concurrent programming and design patterns. Moreover, many mainstream general-purpose languages (GPL) do not support built-in features for implementing protocols.

One way to increase confidence in the correctness of protocol implementations is testing. Conventional testing consists of iteratively supplying input and observing if the output is within expected bounds until confidence is sufficient. Testing also only can show the presence of bugs and not their absence [13]. In contrast to testing, model checking can prove specific properties. It can detect violations of user-specified assertions. One example is the assertion that a program is deadlock-free. Deadlocks are hard to detect with conventional testing. Model checking techniques can provide state-space coverage guarantees and can find bugs that would be hard to find with traditional testing. Model checking is, however, more computationally expensive. The relatively high costs of model checking prevent the broad adoption of model checking [17].

Developers regularly strive to comply with the separation of concern principle. According to this principle, every unit of code should only have one purpose. It makes code easier to test or reuse [12]. In mainstream languages, protocol code is often interleaved with low-level code of thread implementations. For example, imagine two separate threads communicating with each other. The most straightforward way to implement such behavior is by

adding code for communication to each thread. However, doing so is not compliant with the separation of concerns principle and it complicates testing this code for the intended communication behavior (the communication protocol).

To overcome the difficulties of manually programming protocol implementations for concurrent programs, researchers created several domain-specific languages (DSL) for programming protocols in the past decade [2, 21, 48]. The main idea behind such DSLs is that protocols can be defined separately from threads, with high-level syntax and clear and uniform semantics. Protocol specifications written in a DSL are compiled into GPL code to obtain an entire executable concurrent program.

Having protocols defined separately brings another advantage. The protocol specification, written as separate units in DSL code, can be represented as a state machine. We can use this state machine to model check the protocol. This would have been much more difficult when the protocol code is dispersed among the threads' code. Current model checkers rely on an abstraction of the protocol specification. A state machine representation is abstracted from the protocol specification as the input for an external model checker.

With this thesis, we aim to contribute to ongoing research [25] that aims to simplify unit testing of concurrent programs that use channel/queue-based programming abstractions for protocol programming using model checking. We explored a new approach to model checking of protocol code to improve the current model checking techniques. We used an adaptation-based approach to model checking. With adaptation-based model checking there is no need for abstraction techniques to construct abstract models to verify [17]. In short, this thesis answers the following research question:

How can adaptation-based model checking be applied in the context of a domain-specific language for protocols?

To provide evidence for feasibility and the practicality of this concept, we developed and validated a scientific proof of concept implementation named "Prut4j". With Prut4j, we can define protocols and compile them into protocol implementations in Java. We can formulate assertions about the protocol implementations' communication behavior and validate them using a model checking algorithm that uses real executions of the protocol implementations to construct a state-space. We conducted three different case studies to show Prut4j's distinguishing power of model checking, its ability to test complex flows and the ability to integrate the generated protocol implementations in real-world applications. The run-time performance of Prut4j has been tested on Cartesius, the Dutch national supercomputer, and results show little performance decrease (below 5 %) and, in some cases, even a performance increase. A paper based on this thesis is accepted for publication and presentation at ICST 2021 [43].

To give context to subsequent chapters, we first explore various aspects of the problem domain; we will elaborate on those in Chapters 2 and 3. Prut4j will be discussed in Chapters 5, 6, 7, 8, and 9. The case studies are discussed in Chapter 10 and the run-time performance evaluation is discussed in Chapter 11. We draw our conclusions in Chapter 12 and discuss future work in Chapter 13.

I

BACKGROUND

2

DOMAIN SPECIFIC LANGUAGES FOR PROTOCOLS

2.1. DIFFICULTIES IN CONCURRENT PROGRAMMING

Debugging communicating concurrent programs is a critical challenge in concurrent programming [4]. Concurrent programs consist of threads that perform sequential computation and protocols that govern their concurrent interaction [26]. A protocol implementation is responsible for enforcing that only admissible interactions occur. We need to ensure that threads indeed follow the intended protocols. Programmers that are responsible for creating those protocols cope with issues like locks and semaphores [5]. Modern programming languages offer high-level primitives like channels and queues to prevent bugs related to those issues. However, evidence suggests that channel/queue-based programming abstractions have their issues, too. For instance, a study of 171 concurrency bugs in major open-source programs shows: "message passing does not necessarily make multi-threaded programs less error-prone than shared memory" [45].

To illustrate these challenges, we observe the following example. Imagine that we must write a program to simulate a game of chess. We could write a concurrent program where two threads exchange messages over queues with a turn-taking protocol [25]. See Figure 2.1 for a possible Java version of this program. The protocol has two roles, White for the player with the white pieces, and Black for the player with the black pieces. Black simulates a computer player. Every thread has a local copy of the board and runs a loop. In each iteration:

- White receives a move from Black; then updates (its local copy of) the board; then writes the board to the terminal; then reads the next move from the terminal; then updates the board; then sends its move to Black.
- Black pre-analyses (its local copy of) the board for possible next moves (*before* knowing White's move); then receives a move from White; then updates the board; then selects a move based on the pre-analysis (*after* knowing); then updates the board; then sends its move to White.

The main method starts a thread for every role. White's turn is on Lines 7-16, and Black's turn is on Lines 21-28. White is allowed to send a message to Black and vice versa, but only after receiving a message from the other player, except for the first message from White

```
1   public static BlockingQueue wb = ...;
2   public static BlockingQueue bw = ...;

3   public static void runWhite() {
4       Board b = new Board();
5       while (!board.final()) {
6           if (!board.initial()) {
7               Move mBlack = (Move) bw.take();
8               b.update(mBlack);
9               if (board.final()) break;
10          }
11          b.writeTo(System.out);
12          Move mWhite = b.readMoveFrom(System.in);
13          b.update(mWhite);
14          wb.put(mWhite);
15      }
16  }

17  public static void runBlack() {
18      Board b = new Board();
19      while (!board.final()) {
20          b.pre_analyse(); // long-running method call
21          Move mWhite = (Move) wb.take();
22          b.update(mWhite);
23          if (board.final()) break;
24          Move mBlack = b.decide();
25          b.update(mBlack);
26          bw.put(mBlack);
27      }
28  }

29  public static void main(String[] args) {
30      new Thread(() -> runWhite()).start();
31      new Thread(() -> runBlack()).start();
32  }
```

Note: `Board` and `Move` are custom Java classes (details unimportant); `Thread`, `BlockingQueue`, and `System` are Java classes in the standard library.

Figure 2.1: Concurrent chess program (highlights: turn-taking protocol)

(white always starts) to prevent a deadlock. This turn-taking behavior is crucial for the chess game.

Both threads could also have their logic running separately. Black, the computer player, will be running on a background thread and could be calculating possible next moves when waiting for White's move. White, the human player, will interact with some user interface. This user interface will run on its thread while sending and receiving moves in the background. In most popular languages, it is possible to unit test the behavior of the separate threads. For example, we could test for known valid outcomes when testing the next move calculation procedure, or we can check specific interactions with the user interface for a valid outcome. However, imagine testing the turn-taking behavior with conventional unit testing. This is a non-trivial task because current unit test approaches target computations, not communications [46].

The communication-specific behavior in Figure 2.1 is highlighted. These calls to the `put` and `take` methods are send and receive actions over a channel or queue. The combination of these method calls ensures the correct implementation of the turn-taking protocol. Correct use of the `put` and `take` methods is crucial. If they are wrongly used (i.e., in the wrong order), the whole chess simulation would be faulty. It is therefore important to test this protocol code, and it should be easy.

Unfortunately, it is not. The threads Black and White are both responsible for moni-

toring communication with the other thread and are calling the low-level communication primitives (the `put` and `take` methods). These difficulties make it harder to implement both threads and test the communication behavior. The protocol code can only be tested indirectly by testing Black and White threads implementation because protocol code has not been abstracted from the threads and put in a separate module. This 'indirect' testing is disadvantageous because it requires many thread interleavings to be investigated (time-consuming and unsupported by mainstream tools). It is also too imprecise for debugging (if the test fails, it is unclear if the bug is in the protocol code or elsewhere). Sadly, non-isolation is common.

While modularization and separation of concern are important principles in many computing areas, it is not broadly applied for protocols [38, 12]. Dividing software into separate modules makes it manageable, flexible, comprehensible, and reusable. For concurrent programming, this means that all threads and protocols should have a single responsibility, but protocol code often is intertwined in the code of the individual threads [46, 12].

Difficulties in concurrent programming. Programming concurrent programs is a complicated and error-prone task because protocol code often is intertwined with the code of individual threads.

2.2. PROTOCOL PROGRAMMING

Current practices have no well-established development methodologies to implement protocols. DSLs for protocols offer a way to specify protocols, with protocol-specific primitives in a modular way, and can be defined using a textual or visual syntax. When using a DSL for protocols, the complexity of implementing synchronization and communication is not the responsibility of the thread's implementation [46]. This makes integrating protocol programming in current development processes more manageable. In the next section, we will discuss the usefulness of such DSLs.

2.3. SEPARATE PROTOCOL MODULES

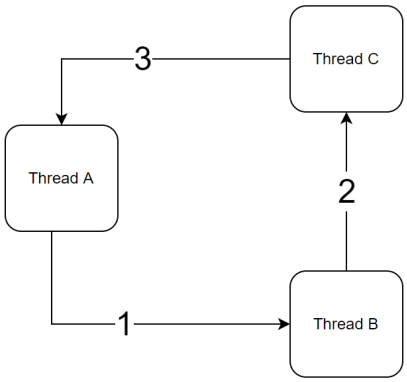

Figure 2.2: Threads in the wild

We will illustrate how we can put communication-specific behavior in separate protocol modules using an example program with three threads, visualized in Figure 2.2. For

this program to function correctly, it must fulfill some requirements regarding communication behavior. Thread A will always start the communication by sending a message to Thread B. Thread B will subsequently send a message to Thread C, and finally, Thread C will send a message to Thread A. To ensure there is no violation of these requirements, we must implement context-aware logic in all threads.

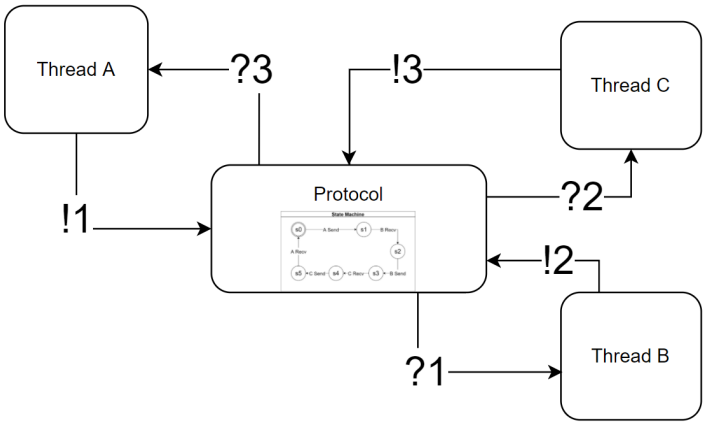

Figure 2.3: Threads with a separated protocol module

Figure 2.3 shows an updated version of the program. Implementing the threads is now simplified since they now only communicate with the protocol module. All threads can be created and tested separately, without the responsibility to ensure correct communication behavior. The protocol module could ensure correct communication behavior by implementing an internal state machine. Figure 2.4 shows the state machine for the example program. States are labeled with a state identifier (S0, S1, ...), and transitions are labeled with the communication action that is possible from that state. S0 is the initial state. From this initial state, only a send operation from thread A is possible. If this operation is executed, the state machine in the protocol module will end up in S1. From S1, there is only a receive action possible from thread B. The state machine in the protocol module will always be in a specific state where only specific behavior is possible.

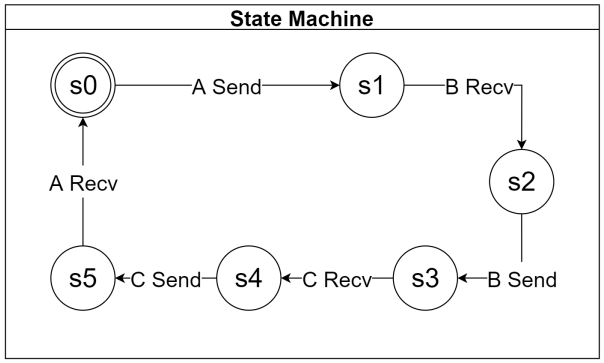

Figure 2.4: A state machine for a protocol module

Creating such a state machine for this simple communication structure in languages like Java can be laborious but still feasible. Now imagine a more complex protocol with more threads and varying communication structures. Figure 2.5 shows a protocol module that should ensure correct communication among multiple threads, where even some

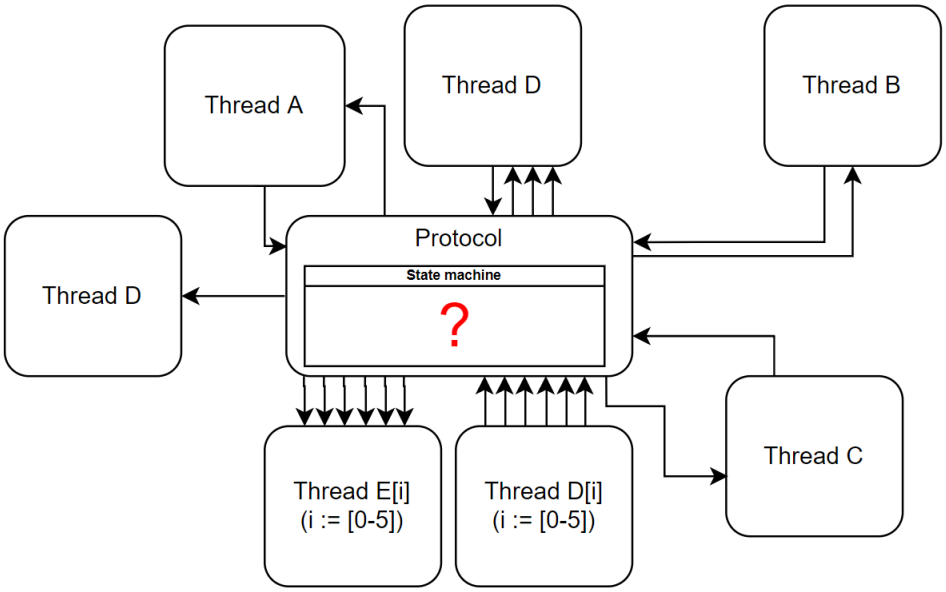

Figure 2.5: Complex concurrent program

threads are parameterized. Writing this implementation would become more complex and error-prone. Separating the protocol code from the threads is not easily done and requires non-trivial discipline.

DSLs for protocol programming solve some of these difficulties. Protocol modules, written in such DSLs, can be compiled into low-level protocol implementations. Such DSLs enforce separation and offer high-level abstractions. Figure 2.6 shows an example protocol definition for the state machine in Figure 2.4 in a fictive DSL.

```
1 PROTOCOL MyProtocolModule {
2     REPEAT {
3         Msg FROM ThreadA TO ThreadB;
4         Msg FROM ThreadB TO ThreadC;
5         Msg FROM ThreadC TO ThreadA;
6     }
7 }
```

Figure 2.6: Protocol definition for the protocol in Figure 2.4 in a fictive DSL.

Relevance of DSLs for protocols. We have shown that manually separating communication behavior into protocol modules is a complex and laborious enterprise. Code generators for DSLs for protocols generate similar protocol code. This shows the relevance of DSLs for protocol programming since writing such implementations by hand can be very laborious and error-prone.

2.4. ARCHITECTURE OF DOMAIN SPECIFIC LANGUAGES FOR PROTOCOL PROGRAMMING

In this section, we will dive deeper into the architecture of DSLs for protocols. Figure 2.7 shows a generic architecture overview of a DSL for protocols. Blue arrows represent manual actions, and green arrows represent automated actions. Developers create threads in GPL code and protocol units in DSL code based on program specifications.

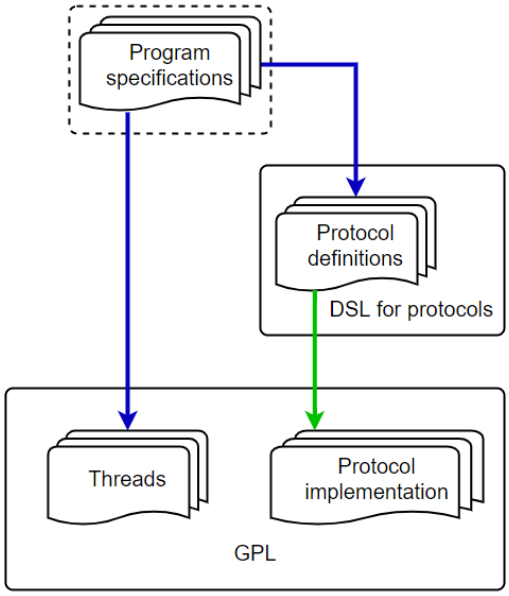

Figure 2.7: General architecture of a DSL for protocols.

DSL code compiles into lower-level protocol code. The final executable program consists of the handwritten threads and the generated protocol code, both in the target GPL (See the box titled 'GPL'). The DSL itself is language-independent. Only the code generators need support for target languages. Extending the support of the code generator with an additional language is relatively straightforward as long as the target language has some form of multi-threading [26].

The DSLs Reo [2, 46] and Scribble [21, 48] are examples of DSLs that use such an architecture. The protocol-specific code is moved from the thread code to a separate unit of code, generated from Reo or Scribble code [46].

Some DSLs use additional intermediate representations before code generation. It is often used for simplicity reasons when generating code and parameterizing the protocol specification when the number of target threads may vary [46].

Architecture of DSLs for protocol programming. Protocol definitions are compiled into protocol modules in a target language. The result is a full program in a GPL. Protocol definitions, written in such a DSL, are only relevant during development.

3

MODEL CHECKING

Clarke et al. defined model checking as follows: "Model checking is an automatic technique for verifying finite-state concurrent systems." [9]. This does not imply that a 'model' of a system is checked. Model checking is a technique to validate some assumption in a temporal logic formula against a system. In the context of model checking protocol implementations, we use the following definition: 'checking whether the graph representing the state-space of the protocol satisfies (is a model of) the property to be checked' [15].

Model checking can be characterized as formal verification based on exhaustive state-space exploration [17]. Model checking enables us to verify certain assertions, generally called properties. Those properties can be classified as safety and liveness properties. Safety properties specify that some bad thing never happens, like 'The system is never in a state where A and B are both true.' Liveness properties specify that some good things eventually happen, like 'Eventually, the system will be in a state where A and B are both false.' A and B could represent predicates over values of variables. The whole state-space is explored to find counterexamples (a trace where the property does not hold) for some formal property. If no counterexamples are found, the property holds for that state-space. Traditional model checking checks properties of communicating finite-state machines. A modeling language defines the state-space of the model as a product of the communicating finite-state machines, usually defined as a directed graph whose nodes are states of the entire system and edges represent state changes. Branching in this graph is due to branching in components or non-determinism due to concurrency (different orderings of actions performed by different components) [17].

We will explain some of the base concepts of model checking with a simple example using a microwave oven state machine (see Figure 3.1). This example is taken and edited from [9].

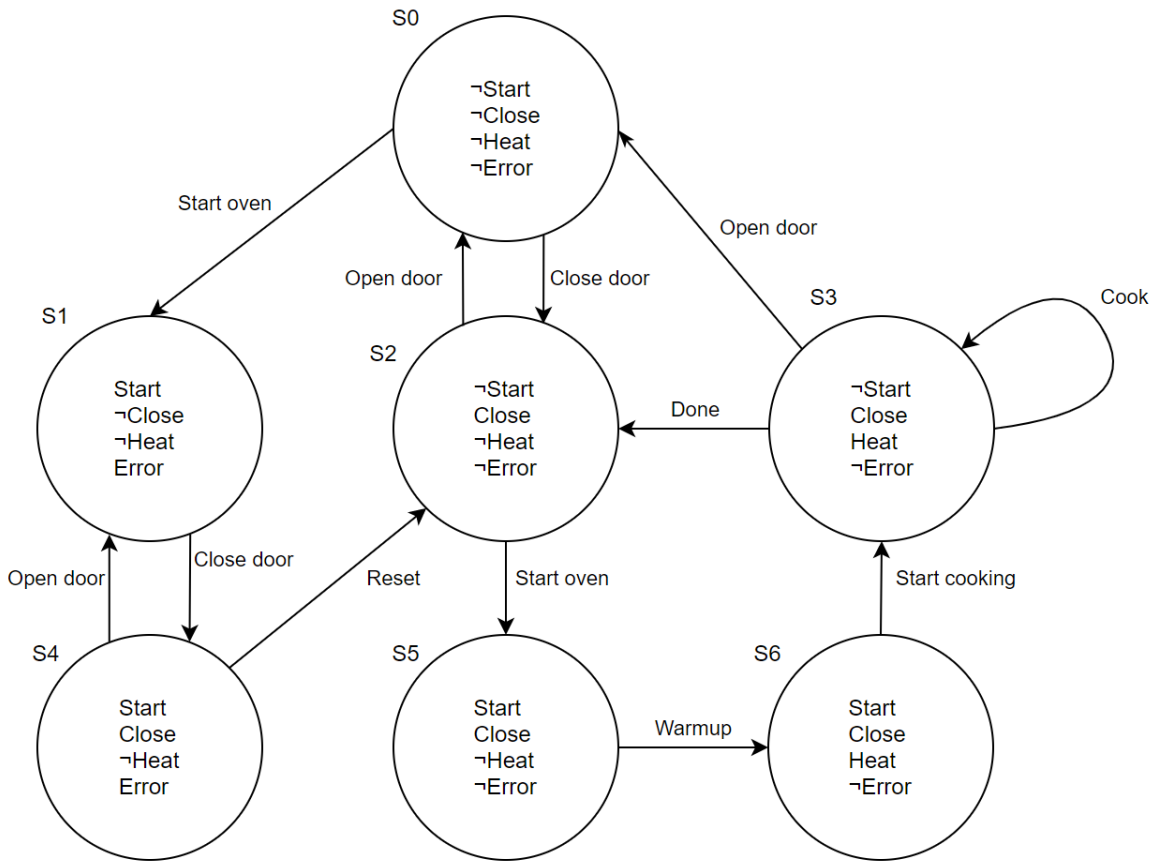

Figure 3.1: The state machine representation of a simple microwave oven.

The state machine in Figure 3.1 shows all possible microwave oven states. For simplicity reasons, we chose to label both the states and the transitions in this example. Usually, either the states or the transitions are labeled since these notations are interchangeable. To guarantee the correct behavior of the microwave oven, one could formulate specific properties of the microwave oven's behavior. These properties could include 'The oven can never be heating and have its door open at the same time', 'It is always possible to open the door', 'The microwave oven always stops cooking when the door is opened' or 'The oven cannot be turned on when there is no food in the oven'. Those properties either hold or not. When a property 'holds' for a specific program, it means that the assertions made in the property are proven to be correct. Sometimes it is necessary to check all states to verify whether a property holds for the model. Other properties are already satisfied when only one specific state is found where the property holds. Exploring all states manually for this small example is feasible. However, this is not the case for most real-world systems with multiple components communicating with each other.

Better coverage. Model checking can be a powerful technique to ensure the correctness of software, and it provides better coverage compared to conventional testing but is more resource-consuming.

3.1. LINEAR TEMPORAL LOGIC

There are various known examples of temporal logic [7, 1, 41, 47]. With temporal logic, we can reason about future paths of state machines. Properties of a system under test can be formally defined in temporal logic. All kinds of scenarios can be defined. Some example scenarios are, 'Condition A must hold in the next state', 'Condition A must hold until Condition B holds,' 'when Condition A holds, then Condition B must hold until Condition C holds.' In the following, we will explain the linear temporal logic (LTL) syntax and relate it to the microwave example with some examples. We chose LTL because of its relatively low complexity and algorithms and optimization techniques for LTL model checking already exist. In Section 7.1.2 we will also argue that LTL is expressive enough for our research. An LTL formula consists of atomic propositions, Boolean operators, and the temporal operators X and U. Formulas can be inductively defined given a finite set of propositions $\mathcal{P}$ as follows [39, 15]:

- All propositions in $\mathcal{P}$ are formulas.
- If φ and ψ are formulas, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $X\varphi$ (next) and $\varphi U \psi$ (until) are also formulas.

$X\varphi$ evaluates to true for all paths where φ holds in the next state. $\varphi U \psi$ evaluates to true for all paths where φ holds until ψ holds. We introduce the following common extensions and abbreviations to the temporal operators:

- $\mathbf{T}$ is an abbreviation for $\varphi \vee \neg\varphi$ or for the Boolean value 'true'.
- $\mathbf{F}$ is an abbreviation for $\neg\mathbf{T}$ or for the Boolean value 'false'.
- Fφ is an abbreviation for $\mathbf{T}U\varphi$, informally: eventually φ will hold.
- Gφ is an abbreviation for $\neg F \neg\varphi$, informally: φ must always hold.
- $\varphi \implies \psi$ is an abbreviation for $\neg\varphi \vee \psi$, informally: ψ should hold when φ holds.
- $\varphi W \psi$ (weak until) is an abbreviation for $\varphi U(\psi \vee G\varphi)$.

With this syntax, we can express informal assertions as formal properties. We will give a few examples of properties for the microwave example:

1. ¬Heat U Close
2. ¬Close U (Start ∧¬Error)

Property 1 holds for the state machine of the microwave example. There is no reachable state that does not satisfy this property. Property 2 does not hold. A counterexample is the path S0, S1, S4. The microwave is now in a state where the door is closed. Thus ¬Close does not hold. Then (Start ∧¬Error) should hold, but that is not the case. The formula does, therefore, not hold for the microwave example state machine.

Temporal logic. With LTL, we can express informal assertions as formal properties. Those properties can define conditions for future paths in a state machine. If those conditions are met for all possible paths of the state machine, the property holds for that state machine.

3.2. MODEL CHECKING ALGORITHMS

Model checking consists of checking whether a state-space satisfies (is a model of) a property. In the context of LTL, model checking consists of checking whether all infinite execution sequences that can be extracted from the state-space graph satisfy the temporal logic formula [15]. In the microwave example, this is the case for property 1, as exemplified in the previous subsection. To check whether a given program or protocol satisfies a property, one could do the following [11]:

- Build the state machine that represents the negation of the property, whose state-space characterizes infinite execution sequences that violate the property.
- Create a state machine of the system under verification.
- Take the product of both state machines.
- Check if the product is not empty. We can check for the non-emptiness of the product with cycle detection algorithms [20, 11].

There are various known variants of model checking algorithms for LTL properties [9, 15]. Most variants are geared towards a specific goal, like performance, expressiveness of properties, or exhaustiveness. One example of a model checking algorithm is presented in [15]. With this algorithm, the protocol state-space may be constructed on-the-fly while checking for the product's emptiness. This makes it possible to detect that properties do not hold by only creating a partial product state machine.

Due to practical reasons, guarantees given by model checking are often limited. Model checking capabilities are often bound by hardware or time constraints. Compromises must be made between preciseness, duration, or the properties that are checked [17].

Non-empty product state machine. Some model checking algorithms require checking whether the product state machine of the LTL formula state machine and the system under test state machine is not empty.

3.3. STATE-SPACE EXPLOSION

The state-space of a program can become exponentially large, as it consists of all threads' behavior, interacting in all possible ways [17, 9]. This exponential growth is called state-space explosion. Proving properties can be extremely time and resource-consuming, or even practically impossible, due to state-space explosion. The state-space explosion is one of the biggest challenges in model checking. It is therefore essential to represent the state-space as compactly as possible. Techniques to tame state-space explosion can be reduction-based or decomposition-based.

Reduction-based techniques focus on reducing the state-space to a feasible size. Among reduction-based techniques is partial-order reduction. This technique is based on the observation that the effects of concurrently executing transitions are often independent of the order of the transitions. This means that the system results in the same state regardless of the order of the transitions [8]. Another reduction-based approach exploits the fact that not all states are reachable. The set of reachable states can be a lot smaller than the whole state-space. State-spaces can be constructed incrementally. All next states are found by

taking all possible transitions from the currently reached states. It reduces the state-space since unreachable states are ignored with this approach [24]. Such an approach is used in [15].

Decomposition-based techniques reduce the verification problem of the original program to verification of subprograms, such that the results can subsequently be combined to draw conclusions about the correctness of the full program.

Exponential growth. Exponential growth of a state-space is called state-space explosion. Techniques exist to reduce or decompose this problem.

3.4. SOFTWARE MODEL CHECKING

A particularly interesting technique for our research is software model checking. It does not require abstract, often manually specified, models of the program but works directly with program implementations. Jhala and Majumdar defined software model checking as follows: "Software model checking is the algorithmic analysis of programs to prove properties of their execution." Software model checking has been successful in specific domains (e.g., call-processing software [6, 16]), but not for general-purpose software. There are two approaches for software model checking: abstraction-based and adaptation-based [17, 24, 18].

3.5. ABSTRACTION-BASED SOFTWARE MODEL CHECKING

Abstraction-based model checking consists of two steps. The first step is using static analysis to extract an abstract model from a program automatically. The second step is to use traditional model checking algorithms on the abstract model. Any counterexamples will be mapped back to the code [17].

3.6. ADAPTATION-BASED MODEL CHECKING

Adaptation-based model checking uses techniques that directly construct a state-space by running a program implementation (instead of analyzing the code to obtain an abstract model). It uses a runtime scheduler to systematically explore and monitor all the states of a concurrent program. A runtime scheduler has to consider non-determinism. Non-determinism in concurrent programs can make state-spaces infinitely large. There are two sources of non-determinism. First, non-deterministic behavior can arise through concurrency (interleaved execution of threads). Second, non-determinism can occur through random input data. The runtime scheduler must account for both types of non-determinism for traces (e.g., counterexamples) to be reproducible. The runtime scheduler must keep track of all visible operations of concurrent threads to create a state-space by running and monitoring threads. This means that a runtime scheduler in practice must be able to trap system calls related to communication (send and receive messages), suspend and resume the execution of threads, and effectively control all threads' scheduling whenever they attempt to communicate with each other. As we will discuss in the next chapter is trapping operating system calls not necessary for every system. This will not be necessary in the case of our research.

Adaptation-based model checking must not be confused with adaptive model checking, where the model is adapted to the program's behavior when a counterexample turns

out to be a false positive [18]. This is, in fact, an abstraction-based approach.

Abstraction-based vs. Adaptation-based. Abstraction-based model checkers use an abstract, in most cases simplified, model of the program under test. Adaptation-based model checkers are integrated into the runtime and control runtime behavior.

3.7. SYSTEMATIC TESTING AND MODEL CHECKING

Substantial research has been conducted on unit testing of concurrent programs (e.g., [34, 40, 10, 23, 37]). Testing techniques like this can be confused with model checking. These "existing automated testing tools for multi-threaded code mainly focus on re-executing existing test cases with different schedules" [44]. Systematic testing of concurrent programs often copes with flakiness. Flakiness is one of the biggest issues when testing concurrent programs. Tests of concurrent programs fail due to non-deterministic behavior because threads can be scheduled in different orders (e.g., [35, 32, 42]).

Software model checking is a form of systematic testing with a runtime scheduler and is not subject to flakiness. With systematic testing, the runtime scheduler tries to force the program to all possible execution paths. Eventually, the whole state-space will be explored. However, the state-space of non-trivial programs is often far too big to explore in full. Therefore, real word algorithms test concurrent systems up to a reasonable depth (e.g., 50 transitions). Those depths are often sufficient to test implementations of communication protocols and other distributed applications. Most protocols are designed so that a few messages are sufficient to exercise most of their functionality, making the exploration of all possible interactions up to tens of messages very valuable. This approach to software model checking has repeatedly been proven to be effective in revealing subtle concurrency-related bugs [17]. In this thesis, we show a technique to abstract communication behavior from a concurrent program into a separate protocol module that is not subject to flakiness.

Flaky. Testing concurrent programs is difficult. A common difficulty is flakiness.

3.8. MODEL CHECKING DSLS FOR PROTOCOLS

We will give a few examples of model checkers for DSLs for protocols. A symbolic model checking approach is described in [28], where a constraint automaton representation is abstracted from Reo syntax and analyzed by the model checker. [27] presents an approach where Reo protocol specifications are converted to Büchi automata that are then used to apply model checking. [3] and [30] present another approach. Both convert Reo syntax to some intermediate language that can be used by external model checkers, like the Vereofy or mCRL2 model checker. An overview of the model checking capabilities of those approaches is given in [29].

Figure 3.2 shows a coarse-grained overview of the characteristics of the approaches mentioned above. The protocol definitions are used to obtain an abstract model of the protocol module. External model checking tools use this abstract model.

Abstractions. Current DSLs for protocol programming use abstraction-based approaches to model checking.

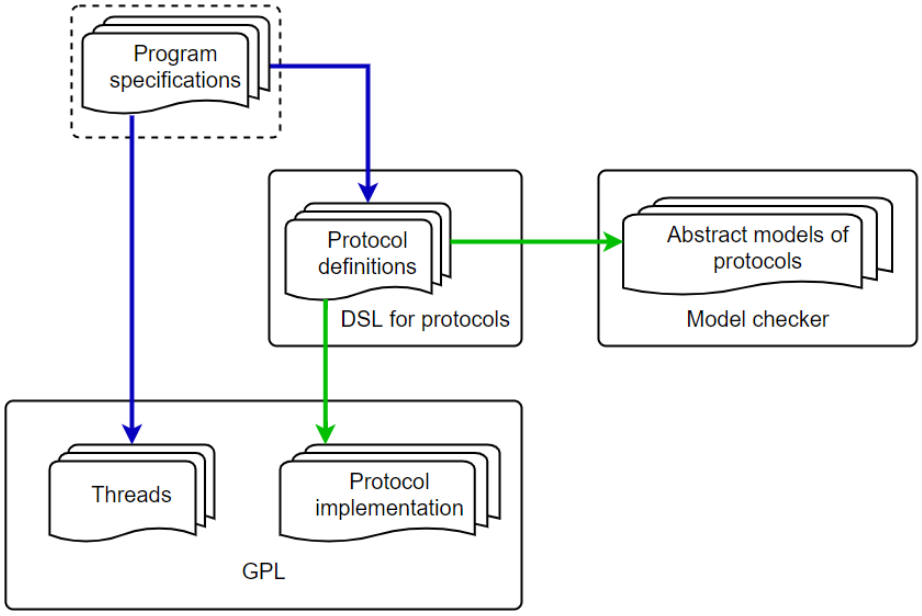

Figure 3.2: General architecture of a DSL for protocols with model checking.

4

ADAPTATION-BASED MODEL CHECKING FOR PROTOCOL PROGRAMMING

As mentioned in the previous chapter, current model checkers for protocol programming use an abstraction-based approach. There is a downside to this approach. All properties that are proven with model checking hold in the state machine representation that was abstracted from the DSL code. However, the DSL code is not the actual running code but is only used for code generation. This means that we need to verify that the proven properties still hold in the generated code. To do this, one needs to find a way to validate the compilation procedure. Validating a code generation algorithm is a labor-intensive task and has to be redone after each change to the algorithm. Instead, we propose a new adaptation-based approach to model checking of protocol code. We use a code generation-based DSL approach so that model checking will be done with the actual generated low-level code.

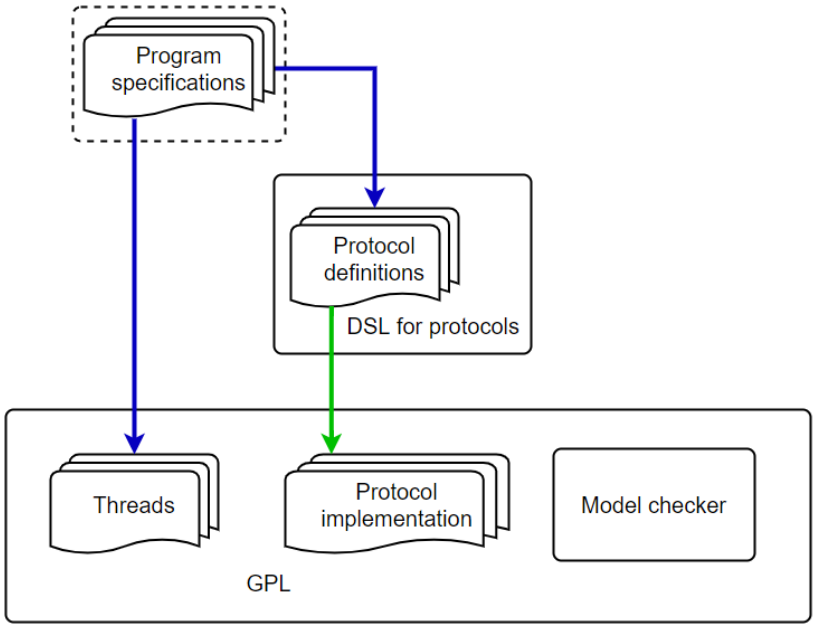

Figure 4.1: New architecture of a DSL for protocols with model checking.

With this new approach, there is no dependency on an external model checking tool.

This could be a disadvantage since external model checking tools can be very powerful. However, removing this dependency could result in easier integration in modern development environments and continuous integration environments. Also, this adaptation-based approach does not exclude abstraction-based techniques. Using external model checking tools with abstractions of the DSL code could still be possible.

II

CONTRIBUTION - PROOF OF CONCEPT

5

PRUT4J

To provide evidence for the adaptation-based approach to model checking protocol code, we created Prut4j. Prut4j is a proof of concept implementation that uses such an approach. Prut4j can be used to write protocol modules and unit tests. Developers will keep writing software in a mainstream GPL (Java in the case of Prut4j), except for protocol modules and protocol unit tests. Protocol modules can be written in *Discourje* [19] and unit test in *Owl* [31] (a DSL based on the LTL syntax). Prut4j is responsible for integration into the low-level GPL code and executions of the unit tests using model checking techniques.

This chapter will give an overview of Prut4j to better understand the context for the following chapters. Then we will discuss the technical choices we made when creating protocol implementations in Chapter 6, the model checker in the runtime in Chapter 7 and the code generator in Chapter 8. In Chapters 10 and 11, we discuss how we validated Prut4j with case studies and performance benchmarks. Examples in this chapter are taken from [43]; Prut4j is presented in this article.

We demonstrate the usage of Prut4j with the chess program from Chapter 2.1. This chess example uses a turn-taking protocol. We will give the protocol definition in Discourje and discuss some unit tests written in Owl. We will show how the two are integrated into the program in Java.

5.1. PR INTERFACE

```
1    public interface Pr {
2        Optional exch(String threadName, Optional box);
3        default void send(String threadName, Object message) {
4            exch(threadName, Optional.of(m));
5        }
6        default Object recv(String threadName) {
7            return exch(threadName, Optional.empty()).get();
8        }
9    }
```

Figure 5.1: The `Pr` interface

Protocol modules that are generated by Prut4j implement one interface, shown in Figure 5.1. The `exch` method needs to be implemented by the implementing class. Note that the `send` and `recv` methods are just wrappers for special usages of `exch`. This `Pr` interface

encapsulates the protocol. A thread can call `send` or `receive` instead of calling the lower-level primitives `put` or `take` of a queue. A thread only calls `send` or `receive` and does not need to monitor when or with whom it is communicating. This logic exists within the implementing class of `Pr`. If a thread calls a `send` or `receive` operation, it will remain blocked until the protocol is in the state where that action is possible. The calling threads do not contain any protocol logic anymore.

5.2. TURN-TAKING PROTOCOL IN DISCOURJE

Discourje is a DSL for channel/queue-based protocols based on *Lisp*, originally developed in the context of runtime verification for protocols. We use Discourje in Prut4j to define protocol definitions and compile them to a state machine. The Discourje compiler takes a `.dcj` file as input and outputs a state machine. Prut4j uses this state machine to generate protocol modules in the form of a Java class. This Java class implements the `Pr` interface and runs, in fact, a state machine.

```
1 (loop [p "White"
2        q "Black"]
3   (-->> Move p q)
4   (recur q p))
```

Figure 5.2: Turn taking protocol in Discourje

Figure 5.2 shows the turn-taking protocol in Discourje and Figure 5.3 shows the state machine of the turn-taking protocol. The labels on the transitions ($[p,q]\dagger m$) indicates a send (if $\dagger$=!) or receive (if $\dagger$=?) of a message of type m through the channel/queue from thread p to thread q.

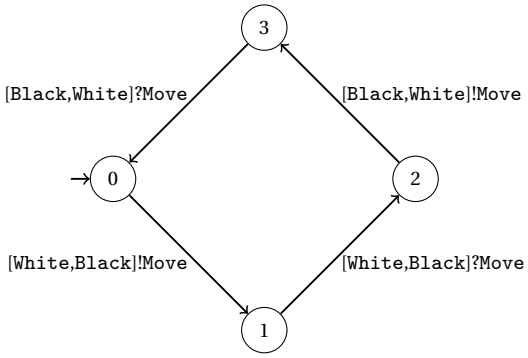

Figure 5.3: Turn taking protocol state machine

This protocol definition defines a parametrized loop. `p` and `q` represent thread names and are given the values `"White"` and `"Black"`. An iteration starts with `p` sending a `Move` to `q`. The `recur` keyword indicates the start of the next iteration. `p` and `q` are swapped here. In other words, the main action in the loop consists of a communication action where a message of type `Move` is sent and subsequently received over a channel between `p` and `q`. Then the turn has been passed, and the same action is possible with the threads swapped.

Timing of `send` or `receive` operations not only relies on the queues but also on the state of the protocol. This enables developers to make more sophisticated communication structures.

Discourje & Prut4j.
The developer writes a protocol in Discourje (Figure 5.2), and then, Prut4j's compiler internally constructs the corresponding state machine (Figure 5.3) and generates a class that implements the `Pr` interface.

5.3. LTL PROPERTIES IN OWL

Owl is a DSL to define requirements for a protocol based on *linear temporal logic*. With Owl we can formally define informal requirements like: "if Black has not received from White, then Black cannot send to White". This requirement is formally defined in `test2.owl`, in Figure 5.4. This figure shows a selection of LTL properties in Owl, stored in files `test[1,2,3].owl`. The property in `test1.owl` states that initially, Black can*not* (`!`) send a message of type `Move`. The second property states that Black cannot send *until* (`U`) Black receives; see example above. The last property states that *eventually* (`F`), *if* Black sends (`=>`), *then next* (`X`), it cannot send again until it has received.

- `test1.owl`

```
!"Black SEND Move"
```

- `test2.owl`

```
!"Black SEND Move" U "Black RECV Move"
```

- `test3.owl`

```
F("Black SEND Move" => X(!"Black SEND Move" U "Black RECV Move"))
```

Figure 5.4: LTL properties in Owl files for the chess protocol.

The Owl component in Prut4j compiles the `.owl` file to a state machine representation of that formula. This state machine is used by the engine class that executes a model checking algorithm to check the formula for a protocol implementation. This engine class is called with the state machine, a `Pr` object and "dummies" as input and returns `true` or `false` after the execution of the model checking algorithm. Dummies are objects that simulate real messages of the protocol. In the case of the turn-taking protocol, we will need to pass a `Move` object.

The engine explores every possible run of the `Pr` object and checks if the formula written in Owl is a model of the protocol module. To explore the possible runs of the protocol module, sequences of calls are done on the `Pr` object in a guided dept-first-like manner.

Figure 5.5 shows a JUnit class that the developer can write for the tests in Figure 5.4. Observe that we defined a dummy object for the Move class. If a `Pr` class contains more types of messages, there should be more dummies.

Owl and JUnit. The developer writes Figure 5.4 and Figure 5.5, and then, Prut4j's execution engine re-constructs the state machine of the `Pr` object and inspects it.

```
1  public class TurnTakingPrTest {
2    private Object[] dummies = new Object[] { new Move() };
3    private Pr p;
4
5    @BeforeEach public void init() {
6      p = new TurnTakingPr();
7    }
8
9    @Test public void testBlackCannotSend() {
10     assertTrue(Engine.exec("test1.owl", p, dummies));
11   }
12
13   @Test public void testBlackCannotSendUntil() {
14     assertTrue(Engine.exec("test2.owl", p, dummies));
15   }
16
17   @Test public void testBlackCannotSendAgainUntil() {
18     assertTrue(Engine.exec("test3.owl", p, dummies));
19   }
20 }
```

Figure 5.5: Selection of unit tests in JUnit ≥5, using Figure 5.4

5.4. INTEGRATION

Using the `Pr` interface from Figure 5.1 in a real program requires little effort. The class from Figure 2.1 can simply be updated by replacing the queues with a `Pr` object and the `put`/`take` calls with `send`/`recv` calls. When the `Pr` interface is used from the start for new programs, no refactoring is required. Integration of the unit tests is also very straightforward. Tests can simply be executed as JUnit tests, see Figure 5.5.

5.5. MANUAL TESTING

```
1      @Test public void testBlackCannotSendUntil_1() {
2          p.send("White", dummies[0]);
3          synchronized (p) {
4              new Thread(() -> {
5                  synchronized (p) {
6                      p.interrupt();
7                  }
8              }).start();
9              assertThrows(InterruptedException.class, () -> p.send("Black", dummies[0]));
10         }
11     }

12     @Test public void testBlackCannotSendUntil_2() {
13         p.send("White", dummies[0]);
14         p.recv("Black");
15         p.send("Black", dummies[0]);
16     }...
```

Figure 5.6: Selection of unit tests in JUnit ≥5, *not* using Figure 5.4 (cf. Figure 5.5)

To illustrate why model checking techniques are powerful in this case, we observe Figure 5.6. It shows that it is possible to manually write JUnit tests without Prut4j's `Engine`. However, the tests shown are incomplete compared to the test based on `test2.owl` in Figures 5.4 and 5.5, and it requires much work to write all the tests by hand. As the state machines for more extensive protocols will grow, writing these test methods will become practically impossible. In other words, it is almost impossible to get the same coverage as Prut4j based tests when creating the tests manually.

5.6. CORE COMPONENTS

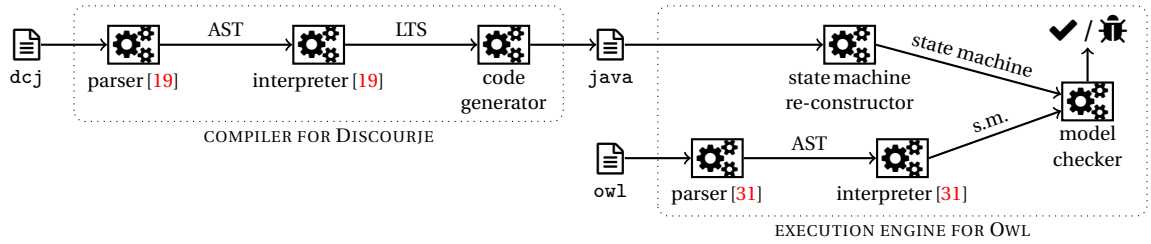

Figure 5.7: Prut4j Architecture

Figure 5.7 shows the core components of Prut4j and the data flow among them. The two main components are the compiler for Discourje and the execution engine for Owl.

The Discourje compiler works as follows:

1. The **input** is a protocol definition in Discourje. The syntax of Discourje is a subset of Clojure extended with macros for protocols (e.g., `-->>` in Figure 6.2). Clojure is a Lisp-based language running on the Java Virtual Machine.

2. The **output** is a Java class that implements interface `Pr`.

3. The **parser** and **interpreter** are both written in Clojure. The parser converts Discourje code into an internal *abstract syntax tree* (AST) by taking advantage of Clojure's own macro expansion mechanism. The interpreter converts the AST into a labeled transition system (LTS) data structure.

4. The **code generator**, written in Java, converts an LTS (generated by the interpreter) into the output Java class.

The parser and interpreter for Discourje were originally developed as part of a library for *runtime verification* in Clojure. Details of this parser and interpreter (including formal semantics) can be found elsewhere [19]. While the parser and interpreter were already present, the code generator has been built for this research.

Executing the model checking engine works as follows:

- The **input** is a `Pr` class and a unit test in Owl. It also requires some configuration data like the dummy objects. Those are omitted in Figure 5.7.

- The **output** is a model checking result: the input formula (from the Owl file) does either hold (the engine returns true) or not (the engine returns false). If a test holds, it holds for every run of the `Pr` class (✔). If a test does not hold, it does not hold for at least one run of the `Pr` class. A bug is reported (🐞), and the engine prints a counterexample trace to the console output. Note that once a counterexample is found, it is directly reported, and the model checking algorithm terminates. It does not report all possible counterexamples.

- The **state machine re-constructor**, creates a graph data structure that represents the same state machine as the LTS in the compiler for Discourje. Specifically, every vertex

v is a `Pr` object, while every edge (v, v') represents a `send/recv` call on v, resulting in v'. Thus, every path $v_1 v_2 \cdots v_n$ through the graph represents a run of v_1. The state machine re-constructor is written in Java.

The re-constructor starts with a new `Pr` object, this is the initial state. All possible `send/recv` operations are tried on clones of this initial state. If an operation is possible, then we found a successor state. The whole state-space is explored by the model checker in a guided depth-first search manner.

- The Owl **parser** and **interpreter** are written in Java. They parse the property from an Owl file into an internal AST and the AST into a state machine. This state machine represents all runs that are allowed according to the formula in the Owl file. We will call this state machine the LTL state machine since the supported syntax is based on *linear temporal logic* (LTL) [39]. The parser and interpreter for Owl come from an advanced library for temporal logic [31].

- The **model checker** is written in Java. The model checker uses the LTL state machine and the re-constructor to compare the two state machines. An algorithm checks if every run of the re-constructed state machine is also a run of the LTL state machine. If no run is found that is not also a run of the LTL state machine, the LTL property is satisfied. This is implemented efficiently with an algorithm based on *nested depth-first search* [11], using *on-the-fly* techniques [15].

The Discourje parser, Discourje interpreter, Owl parser and Owl interpreter are created by other researchers. Some small changes are made to make it compatible with Prut4j. In contrast, the code generator, state machine re-constructor and model checker are implemented from scratch for this research.

If we look at the chosen architecture, it also seems possible to directly use the LTS data structure (output from the Discourje interpreter) while executing a model checking algorithm. There would not be a need for state-space reconstruction in that case. This is, however, an important decision in our work. We chose to use state-space reconstruction for a couple of reasons:

- The LTS data structure is not the code that is actually running.

- The code generator could produce `Pr` classes that differ from the LTS. This could be because of optimization (actually used in this research, see Section 6.2.3) or bugs in the code generator. When the `Pr` class differs from the LTS, and the LTS is used when model checking, the verdict could be faulty.

Therefore it is essential to reconstruct the state-space with the same code that is normally executed when running programs in the target environment.

6

PROTOCOL IMPLEMENTATIONS

6.1. DESIGN

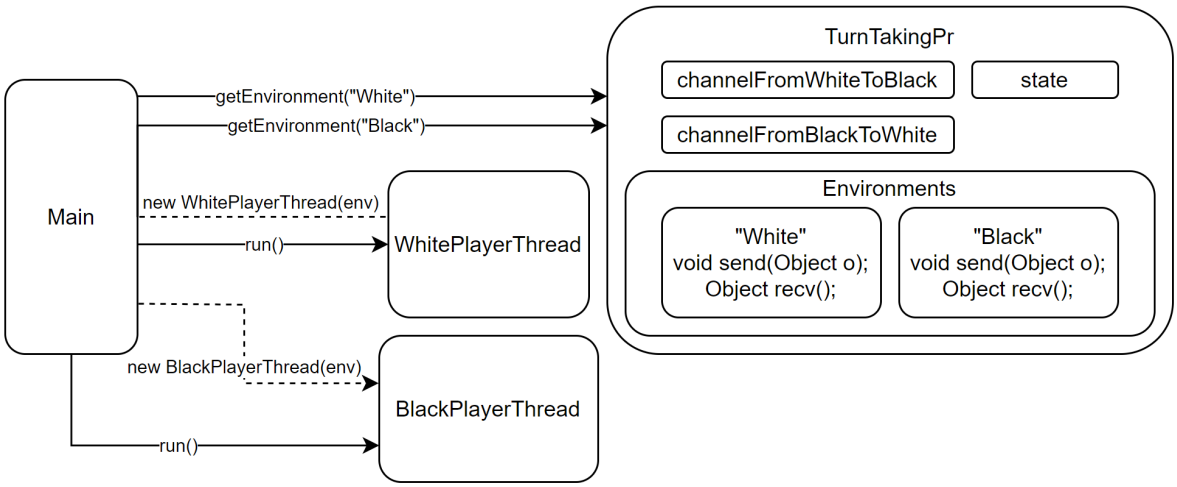

Figure 6.1: Chess protocol in a separated module

Figure 6.1 shows how we can put the turn-taking protocol in a separated protocol module. The `TurnTakingPr` class contains channels for communication. For Chess, these channels are for communication from Black to White and from White to Black. The `TurnTakingPr` class does have a list of environments and a `state`. The program is started in the main component (often the `main` method) that starts the concurrent threads for roles "Black" and "White". When starting a thread, it obtains the corresponding environment for that role and passes it to the thread. Black is started with the "Black" environment and White with the "White" environment. For every possible value of `state`, the environments dictate possible communications for their respective role. Together they implicitly define the global state machine (visually represented in Figure 5.3). If all participants (threads that enact specific roles) only communicate through their environment, protocol compliance is guaranteed.

6.2. IMPLEMENTATION

6.2.1. EXAMPLE 1: CHESS PROTOCOL

```
1   public class TurnTakingPr implements Pr {
2       private BlockingQueue channelFromBlackToWhite = ..., channelFromWhiteToBlack = ...;
3       private volatile int state = 0;

4       public exch(String threadName, Optional box) {
5           switch (threadName) {
6               case "White":
7                   synchronized (this) {
8                       while (true) {
9                           switch (state) {
10                              case 0:
11                                  if (box.isPresent()) {
12                                      channelFromWhiteToBlack.put(box.get()); // send
13                                      state = 1;
14                                      notifyAll();
15                                      return new Optional.empty();
16                                  }
17                              case 3:
18                                  if (box.isEmpty()) {
19                                      Object m = channelFromBlackToWhite.take(); // receive
20                                      state = 0;
21                                      notifyAll();
22                                      return new Optional.of(m);
23                                  }
24                          }
25                          wait();
26                      }
27                  }
28              case "Black": ...
29          }
30      }
```

Figure 6.2: Turn taking protocol in Java ≥8, generated for the protocol module in Figure 5.2

We will demonstrate how to move the protocol code of the chess example to a protocol module. This protocol must enforce the turn-taking of the chess game. Figure 6.2 shows the Java class (slightly simplified) for Figure 5.2. We will discuss some details of this protocol implementation:

- **No direct access to queues.** Queues are modeled as private fields on the protocol class, see Line 2. Interaction with the queue must now only occur through the `exch` method.
- **The `send` and `recv` methods have a default implementation in the `Pr` interface.** They rely on the implementation of the `exch` method. They fill the `box` parameter. The `box` parameter is empty for receive actions and contains an object-to-send for send actions.
- **The `exch` method is responsible for allowing communication.** It executes an underlying state machine, as shown in Figure 5.3. A call to the method results in a send or receive action when enabled in the current state. It will wait otherwise (see Line 25). Observe that a send action is allowed in state 0 and a receive action in state 3.

To simplify the example, we directly communicate through the `exch` method on the `TurnTakingPr` class. In the real implementation, communication goes through an extra abstraction layer. An environment object is used for every `threadName`.

Possible executions of the protocol class for “White” are as follows:

- `state` **is** 0**, and "White" calls** `send`**.** This action is directly enabled (`box.isPresent()` is true on Line 11). The message is put in the queue (Line 12), the state is updated (Line 13), and the other threads are notified if they are in a waiting state (Line 14). An empty result is returned.
- `state` **is** 3**, and "White" calls** `recv`**.** This action is directly enabled (`box.isEmpty()` is true on Line 18). A message is obtained from the queue (Line 19), the state is updated (Line 20), and the other threads are notified if they are in a waiting state (Line 21). The message is returned, wrapped in an `Optional` (Line 22).
- `state` **is** 0**, and "White" calls** `recv`**.** This action is not enabled, (`box.isPresent()` is false on Line 11). The thread will be put in a waiting state (Line 25).
- `state` **is** 3**, and "White" calls** `send`**.** This action is not enabled, (`box.isEmpty()` is false on Line 18). The thread will be put in a waiting state (Line 25).
- `state` **is not** 3 **or** 0**.** There is no action enabled for "White". The thread will be put in a waiting state (Line 25).

When "White" ends up in a waiting state, it can only continue after a call to `notifyAll()` from another thread ("Black" in this case).

A participant interacts with their environment to send or receive messages.

The `exch` methods of all participants are synchronized and represent one global state machine. This state machine ensures that only allowed actions occur.

6.2.2. DETERMINISTIC DETERMINATION OF THE RECEIVER

We needed to determine if we needed to support non-deterministic behavior in the protocol implementations since it has implications on how we generate and test protocol implementations. To be able to argue about determinism, we observe a protocol with non-deterministic behavior. Its global state machine is visualized in Figure 6.3.

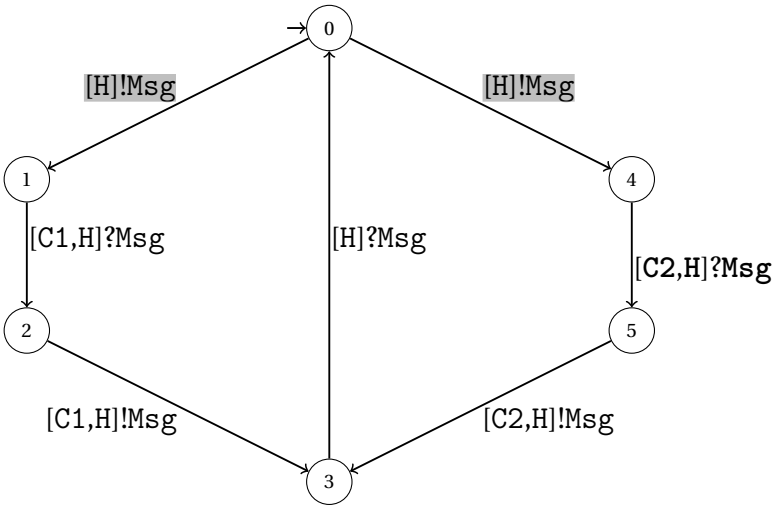

Figure 6.3: The state machine for a protocol with multiple receivers

The state machine in Figure 6.3 shows the communication rules between a human player (`H`), and two computer players (`C1` or `C2`). Note that the receiver can be left out, see transition [0→1] and [0→4]. In this protocol, `H` can send a message, to `C1` or `C2` (transitions `[0→1]` or `[0→4]`), that is received by `C1` or `C2` (transition `[1→2]` or `[4→5]` respectively). `C1` or `C2` will send a response message (transition `[2→3]` or `[5→3]` respectively), that is received by `H` (transition `[3→0]`). There is non-determinism from state 0. When `H` sends a message, either the transition `[0→1]` or `[0→4]` (highlighted) could be taken. We assume that picking a transition is random.

From a developer's perspective, it is practical not to specify a receiver when sending a message. Therefore, we randomly pick a receiver when multiple receivers are possible so that threads can interact with the protocol implementation without knowing its receivers. However, this raises an implementation issue. Assume we want to model check the protocol implementation and check the following LTL property: `G("H SENDS Message"` $\implies$ `¬X("C2 RECEIVES message"))`. This property means informally: It is always true that if the human player sends a message, the next action is not a receive action of the message by computer player two. In other words, computer player two will never receive a message from the human player.

If we want to prove this property, we need to force executions of the protocol through all possible paths to provide full coverage of the protocol implementations state-space. However, we just decided that picking a receiver is random. Assume we are exploring the state-space from state 0, what transition we will take next is random. This means there is a reasonable chance the transition [0→4] (and thus also transition [4→5]) will not be taken for a couple of runs. The more runs of the protocol implementation, the smaller this chance gets. Let us assume that we did five runs and did not detect any `receive` action from computer player two after a `send` actions from the human player (the protocol implementation always picked computer player one as the receiver). We cannot draw the conclusion that our property holds. Even if we would run it a thousand, a million, or even infinite times, there would still be a chance that we will never detect a counterexample (in this case, a counterexample is a path containing transition [4→5]). All guarantees are purely empirical.

```
1  switch (state){
2    case 0:
3      if (box.isPresent() && box.get().getClass() == Boolean.class ) {
4        if (receiver == null) {
5          int rnd = new Random().nextInt(2);
6          String[] receiverOptionsArray = new String[] "worker_1_","worker_2_" ;
7          receiver = receiverOptionsArray[rnd];
8        }
9        if (receiver.equals("worker_1_")) {
10         setState(1); // setState will update the state and call notifyAll().
11         worker_1_Queue.put(new ProtocolMessage(box.get(),1));
12         return Optional.empty();
13       }
14       if (receiver.equals("worker_2_")) {
15         setState(3);
16         worker_2_Queue.put(new ProtocolMessage(box.get(),2));
17         return Optional.empty();
18       }
19     }
20     ...
21 }
```

Figure 6.4: Possible actions from a state with multiple possible receivers.

To counter this problem, we added an optional parameter to the `exch` method, the `receiver`. Threads can interact using the protocol implementation without specifying a receiver while it is still possible to force executions through all possible paths. Figure 6.4 shows a state where sending a message of type `Boolean` is possible to two receivers, `worker_0_` and `worker_1_`. If this `receiver` variable is left empty, the receiver will be picked randomly (Line 4-8).

6.2.3. OPTIMIZATION: LOCAL PROJECTIONS OF THE GLOBAL STATE MACHINE

While validating our proof of concept's run-time performance (discussed in detail in Section 11.2), we encountered a significant performance decrease for several of our generated protocol implementations. To alleviate this, we experimented with a possible optimization. The issue with our base implementation is the shared state. Only one participant can send or receive at the same time. Imagine a protocol with three roles, A, B, and C. The state machine of the protocol is given in Figure 6.5.

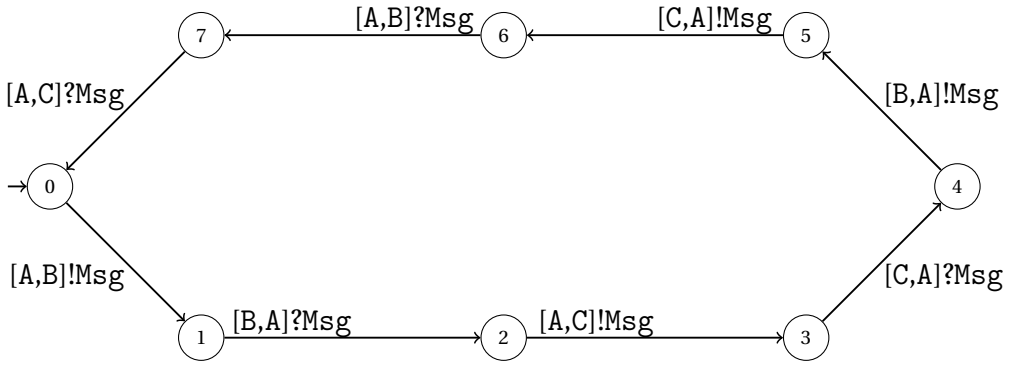

Figure 6.5: The global state machine for a strict protocol definition.

Observe that role A is allowed to send two messages (transitions [0→1] and [2→3]) but has to wait until B has received the message (transition [1→2]) before sending a message to C. The same goes for when B sends a message to A. A will have to wait to receive this message until C sends a message to A. Because of the full enforcement of the global state machine, we call this base implementation 'strict.' The strict behavior is enforced in Java through synchronizations (`synchronized (this) { ... }` on Line 7 in Figure 6.2). This is an inefficient approach when the number of threads grows. The synchronizations have an increasingly negative impact on the performance of the protocol implementations. We implemented a more 'liberal' variant where we use multiple smaller state machines instead of a single one to fix this issue. We decompose the global state machine into a smaller state machine for every thread. In the literature, such decomposition is also called projection (e.g., [22]). The local state machines of every thread are not synchronized and are therefore not blocking.

We created an algorithm to generate a local projection with local states and transitions from a given global state machine. A local transition is a transition in the global state machine where the participant can send or receive a message. Note that transitions, where other participants send a message to the local participant, are not local. A local state will be created by the algorithm and can combine multiple states from the global state machine.

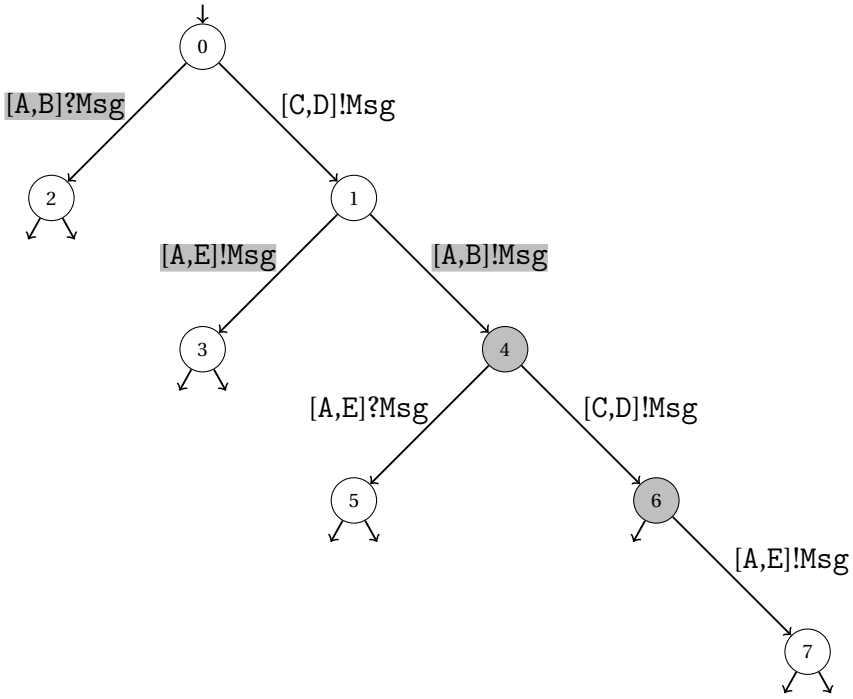

Figure 6.6: A part of a global state machine

The first step is determining a local starting state. Figure 6.6 shows an example global state machine. Assume we are projecting the local state machine for participant `A` and state `0` is the initial state. We travel depth-first through the global state machine and identify local transitions(highlighted) that are reachable by only taking non-local transitions. We now identified the set of outgoing local transitions for the initial local state and can start creating the local state machine (see Figure 6.7). This procedure is applied recursively for all states.

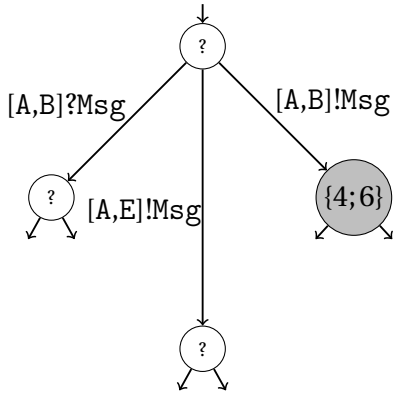

Figure 6.7: A part of a local state machine during projection

The second step is to determine the target states of the outgoing local transitions. All target states that are reachable by only traveling non-local transitions and have at least one outgoing local transition are combined into a new local target state. In Figure 6.6 this is done for state `4`. States `4` and `6` (the gray states) are found and combined in the new local state in Figure 6.7. This procedure is executed for all target states of outgoing transitions and applied recursively until the final local state machine is complete.

Projections to local state machines from the global state machine are made during the generation of the protocol implementation, see Section 8.2. To implement this, we had to change some logic of the `Pr` class:

- **One queue per role.** When using local states, multiple receive actions may be possible in that state. To use the blocking behavior of the Java `BlockingQueue`, we can only use one queue. Otherwise, we would need to monitor all the queues for a new message.
- **Messages over the queue are wrapped in a ProtocolMessage.** Since we are only using one queue per role, we do not know who the sender of a message was (note that a different sender could mean a different branch in the global state machine). For this reason, we chose to wrap a message in the `ProtocolMessage` class where we added the `OriginalTargetStateId` field. This is the target state id of the original global state machine. With this state from the original global state machine available in a message, we know we are not choosing a local action meant for a different branch in the global state machine.
- **Environments have their own state id and initial state id.** Every environment has its own state.

This liberal implementation showed a significant performance increase. This will be discussed in Section 11.2.

Local projections. In optimized liberal protocol implementations, the exchange methods of all participants are not synchronized and contain only local projections of the global state machine.

In Section 3.3, we discussed some techniques to alleviate state-space explosion. This local projection optimization technique can be categorized as a decomposition-based optimization technique. A downside to such decomposition is that some state machines are not possible to decompose with full preservation of semantics (e.g., [22]). This could be a problem since we are trying to ensure the correct behavior of protocol modules and illustrates the importance of adaptation-based model checking of these protocol modules. The underlying state machine will be reconstructed from the real implementation (decomposed or not), and the model checking results are valid for this state machine. If the tests (assertions that the model checker will validate) succeed, then we can conclude that a decomposed protocol implementation is semantics preserving for those scenarios.

Semantics preserving. Some global state machines cannot be decomposed with full preservation of semantics. Adaptation-based model checking can guarantee semantics preservation for certain scenarios.

7

Model checking algorithm

7.1. Design

7.1.1. State-space exploration for protocol implementations

In the previous section, we described how to create protocol implementations that are separated from the threads themselves. Before thinking about generating this code from a DSL, we must first think about how we will use the protocol implementation to reconstruct a state-space for model checking.

To this end, we analyzed protocol implementations we discussed in Chapter 6, and we needed to solve the following problems for the protocol implementations:

- How to detect if we have already been in this state.
- How to detect if we already took the transition before.

This problem's solution will later be useful when creating the product state machine of the protocol state machine and the LTL state machine.

Problem 1, how to detect if we have already been in this state.

Literature describes the use of hash functions (e.g., [11]) to determine already visited states. We do not have to worry about this overhead, and we are able to use a set of identifiers of already visited states. This is not a problem for now since we do not have many distinct states, and the state id of the global state machine (or the combination of the state ids of all local state machines in the case of a liberal protocol implementation) is known.

Problem 2, how to detect if we already took the transition before.

Transitions are labeled with the send or receive message but do not have a unique identifier. The same message could be sent from A to B in multiple places in the protocol. However, since we use a hard-coded (and later generated) protocol implementation, we can easily identify every unique transition using the source and target state identifiers.

State-space explorers

We must be able to explore all possible executions of a protocol implementation so we can fully construct the state-space. Manually programming participants to explore the whole state-space might be easy for the turn-taking example (only a few deviations from

the protocol possible) but could rapidly become more complicated and practically impossible. Therefore we needed to create a state-space explorer. This explorer should be able to interact with the environment in a configured or calculated manner.

We identify two possible approaches:

- **Explorers with calculated paths.** We could generate a set of explorers that perform one possible run of the protocol module. We would need to generate all possible variations to test all executions of the protocol implementation.

- **Explorers that can execute a given action.** Those explorers execute a given action on the protocol object. A send action to a specific role or a receive action. The explorer tries to execute this action on the protocol object and must detect whether the action is actually enabled. This explorer needs to be orchestrated by an external tool or algorithm.

Another downside to the approach with a calculated path is that an explorer can only travel one trace. If we want to travel a trace that only differs after taking multiple transitions, we need a new explorer that travels the whole trace again from the initial state. An explorer who is traveling the protocol state-space must be aware of previously taken transitions by earlier runs to do a full search. This will result in significant memory usage.

The latter is a more practical option in our case. A model checking algorithm that reconstructs the state-space in an on-the-fly manner could use the explorer to produce guided searches over the protocol implementation state machine. Since the protocol object has a state, we need a way to reset or recreate that state. Otherwise, we would need to re-execute the algorithm from the initial state every time we want to explore a different state-space branch.

CLONING PROTOCOL OBJECTS

One way to reset the state of a protocol object is by making copies of the object. Creating manual copies of a protocol object requires extra logic and could be error-prone. To ensure that the copies of a protocol represent the same state, we chose to use reflection-based deep cloning. Using this technique, we ensure that a protocol is copied correctly, even when the internals of a protocol change (e.g., decomposition-based optimizations). If a clone is used to attempt a transition and it turns out to be not enabled, the clone is disposed, and its memory is released. When exploring sub-traces, we do not have to restart execution from the initial state. We create clones of the current protocol object and try to explore different paths. This makes the implementation of a search with cycle detection (the model checking algorithm) significantly less complex.

7.1.2. LTL

If we want to add model checking capabilities to the runtime, we must select a way to express a protocol implementation's properties. Before researching model checking algorithms, we first needed to determine a formal syntax to express these properties. We chose to use LTL for a couple of reasons:

- **Available knowledge.** The author already knew LTL. Studying other temporal logics would cost more time. A thorough evaluation of the applicability of different temporal logics is therefore beyond the scope of this thesis.

- **Expressiveness.** We deem LTL expressive enough for our research. We are able to define useful properties for protocol implementations, as the case studies in Chapter 10 will show. We understand that other formal languages are even more expressive and, therefore perhaps more valuable. Analyzing those possibilities is outside the scope of this research and could be a subject for further research, see Chapter 13.

- **Availability of algorithms.** Algorithms are already available in the literature, see Section 7.1.3.

Formal properties in LTL. We chose to express properties of protocol implementations in LTL

FORMAL PROPERTIES AND STATE-SPACE EXPLORERS

We must express communication characteristics in LTL as atomic propositions, and we must detect them in protocol implementations with the state-space explorers. We must define and detect the participant, the sender or receiver, the message type, and the action (send or receive). We also determined that it is useful to support wildcards in the LTL expressions to make more dynamic expressions.

7.1.3. ALGORITHM SELECTION

We chose to implement an LTL model checking algorithm based on [11]. This algorithm constructs a state-space on-the-fly in the form of a guided search over execution paths. We can use state-space explorers to do this guided search using the protocol object (using clones of this object). To search the possible executions, we will first need the LTL state machine, the state machine representation of an LTL formula. We can reconstruct a state-space on-the-fly while creating the product state machine (the product of the LTL state machine and the protocol state machine) and detecting cycles. The LTL state-machine can guide the search algorithm that orchestrates the explorers. Details of this algorithm will be discussed in Section 7.4.

This algorithm is selected for:

- **The on-the-fly characteristics.** We can check for properties while reconstructing the state-space. For many properties, we will never need to reconstruct the full state-space.

- **State-space explosion friendly.** No full state-space needs to be reconstructed and stored in memory because of the on-the-fly characteristics before executing the model checking algorithm.

- **Simple integration possibilities into the runtime environment.** We can build a search algorithm over possible executions by orchestrating state-space explorers. The LTL state machine can be used to guide the executions.

LTL Algorithm. We chose to implement an LTL model checking algorithm where we construct the state-space on-the-fly while checking LTL properties.

7.2. IMPLEMENTATION

Discussing the implementation of the model checker is challenging to do at once. We divided the main problem into smaller problems, and we will discuss them separately. We will discuss the added state-space exploration capabilities in Section 7.2.2. To construct the protocol state-space in a guided way, we need a state machine representation of the LTL formula under test. This will be discussed in Section 7.3. Finally, we will discuss the algorithm in Section 7.4.

7.2.1. EXECUTION ENGINE

We created an `Engine` class that will execute the Owl unit tests by model checking protocol implementations.

To execute the test, a developer needs to pass the following parameters.

- Path to an `.owl` file that contains an LTL property in the Owl syntax.
- Mapping of short class names to fully qualified names.
- A list of dummy objects.
- A protocol object.

The Engine will try to detect counterexamples of the Owl formula by taking the negation of the formula and executing an algorithm to find traces that satisfy the negated formula. If no counterexamples are found, the formula does hold.

The Engine takes the following steps to execute the unit test:

- Get the content from the `.owl` file.
- Replace short class names using the map.
- Load the Owl formula using the Owl library.
- Negate the formula.
- Try to find counterexamples by executing a model checking algorithm to find accepting cycles.

7.2.2. STATE-SPACE EXPLORERS

The first step towards model checking is creating the state-space explorers that can explore the whole state-space. Once we can explore the whole state-space, we can verify temporal properties using a model checking algorithm. We will discuss the state-space exploration approach in this section.

We assume we have a set of all available possible actions. We created a `StateSpaceExploringThread` to execute a given action on a given protocol copy for a given role. Which action is executed on which protocol copy, for which role, and in which order is up to the model checking algorithm. This will be discussed in Section 7.4. The `StateSpaceExploringAction` class is shown in Figure 7.1.

Figure 7.2 shows the `StateSpaceExploringThread` class. The algorithm creates an instance of this class for every participant. When executing a guided search, the algorithm sets new copies of the protocol object through the `SetProtocolClone` method (See Line 10) and tries to execute a given action by calling the `ExecuteAction` (See Line 23).

```
public class StateSpaceExploringAction {
    public final String participant;
    public final LtlTransitionExpressionAtomicPropositionDirection direction; // Send or Receive
    public final Object dummy;
    public final Class<?> messageClass;
    public final String receiver;

    private StateSpaceExploringAction( ... ){
        // Initialization of all fields
    }
    ...
}
```

Figure 7.1: StateSpaceExploringAction class

```
public class StateSpaceExploringThread {
    private String threadName;
    private Pr protocol;
    private IEnvironment environment;

    public StateSpaceExploringThread(String participantName){
        this.threadName = participantName;
    }

    public void SetProtocolClone(Pr protocolClone) throws Exception {
        // Pass new environment to thread.
        this.protocol = protocolClone;
        this.environment = this.protocol.getEnvironment(this.threadName);
    }

    /**
     * @param actionToBeExecuted Action to be executed represents
     * a possible transition on the protocol state machine
     * @return Will return the protocol after a successful action, is empty otherwise.
     *
     * This method tries a transition, if it is interrupted it has encountered a wait operation.
     */
    public Optional<Pr> ExecuteAction(StateSpaceExploringAction actionToBeExecuted) {
         if (Engine.IsProtocolOptimized) {
            ...
        }else{
            ...
        }
    }
}
```

Figure 7.2: StateSpaceExploringThread class

7.2.3. DETECTION TRANSITION IN THE STATE MACHINE

Figure 7.3 shows how we check whether the `StateSpaceExploringAction` is possible for strict protocol implementations. We must detect whether a send or receive call succeeds. We encounter a problem when the current action is not enabled in the underlying state machine (a wait-action in the `exch` method). This wait-action blocks the exploring thread. There are no other threads that modify the protocol's state, so the wait action is infinite. We could solve this by orchestrating other threads that perform other actions to remove the current exploring thread's blockage. However, this could be problematic because it is subject to non-deterministic scheduling, and other exploring threads could also get stuck (deadlock). This is something we would not be able to detect.

We chose a more sophisticated solution. We use a main thread, an auxiliary thread, and an intrinsic lock within the exploring thread. The lock is set via Java's `synchronized` keyword. The code of the exploring thread class is shown in Figure 7.3. Executing an exploring

action consists of two steps:

1. The main thread acquires the lock (See Line 5) starts the auxiliary thread (See Line 7) and executes the exploring action (See Line 15- 34).

2. The auxiliary thread acquires the lock (See Line 8) and tries to interrupt the main thread (See Line 9). If the lock is acquired successfully, it will interrupt the main thread.

There are two possible outcomes of the execution of an exploring action:

- If the exploring action is possible for the protocol object, the action (send or receive) will be executed directly. The call will return without blocking in a wait action. The main thread will release the lock and wait for the auxiliary thread to finish. The auxiliary thread will acquire the lock and will interrupt the main thread. This interrupt will be ignored. The main thread already returned a value.

- If the exploring action is not possible for the protocol object, the action (send or receive) will not be executed directly. The protocol's state machine will end up in a wait action, blocking the calling thread (the main thread in this case). In a wait action, `wait()` is called. This temporarily releases the lock that the main thread has acquired. Java's intrinsic locks work this way. Then the auxiliary thread will acquire the lock and will interrupt the main thread. The main thread's call to the `wait()` method will terminate with an `InterruptedException`. This exception will be caught (See Line 36) and indicates that the exploring action is impossible for that protocol object. No successor state is found.

Note that we could simplify this procedure by adding a non-blocking version of the `exch` method. However, this violated one of the core principles of this research. We would not be testing the code that is executed during normal run-time.

STRICT VS. LIBERAL PROTOCOL IMPLEMENTATIONS

Liberal protocol implementations require slightly different logic for transition detection. Those protocol implementations will not always end up in a wait state for send operations, so detecting the `InterruptedException` is not always possible but also not necessary. If a send operation is executed while the environment is in a local state where no send operations are possible, a `NotAllowedTransitionException` is thrown. This means that there is no need for the auxiliary nested thread to detect send transitions. The full `StateSpaceExploringThread` class can be found in the supplemental materials, see Appendix A.

State-space exploration. To reconstruct a state-space on-the-fly from a protocol implementation, we created a state-space explorer that the model checking algorithm can use. This explorer takes a protocol copy and an exploring action as input and returns true if this action is possible, false if not.

7.2.4. CLONING PROTOCOL OBJECTS.

To explore the state-space with a search of the protocol implementation executions, we need to create copies of the protocol objects. We chose to do a deep clone of the protocol

object. With a deep clone, we can be sure that the copy is the same as the original object. We chose to use the `kostaskougios:cloning` library to clone the protocol objects for a couple of reasons[1]:

- **It uses reflection to clone objects.** It recursively clones a full object using reflection. Therefore we do not have to worry about object cloning when changing the protocol implementation.
- **Writing it ourselves costs time.** We chose this of-the-shelf library to save time. Writing it ourselves could have been possible but would have cost more time.

7.3. LTL FORMULA TO STATE MACHINE REPRESENTATION

The model checking algorithm we implemented is based on a guided search over possible protocol implementation executions. We have already shown how we can detect possible transitions in the protocol state machine, the `StateSpaceExploringActions` that are tried from the `StateSpaceExploringThread`. We do not want to explore the whole state-space before starting the model checking algorithm, but we want to do this on-the-fly. Therefore we need to conduct a guided search over the protocol state machine. To guide this search, we need the state machine representing the LTL formula that we are checking. To save time, we chose not to create a DSL and a parser but to use an existing one. To this aim, we selected Owl [31], a DSL based on LTL.

We chose Owl for a couple of reasons:

- **Feature rich.** We can express complex LTL properties in Owl and the Owl parser can generate the LTL state machines.
- **Actively developed.** Owl is actively developed and used by the research community [31].[2]
- **Scientifically backed.** Researchers have provided evidence of its correctness [31].
- **Publicly available.** The Owl source code is publicly available.

7.3.1. OWL SYNTAX

The Owl syntax is very similar to the LTL syntax we discussed in Section 7.1.2. We will summarize the syntax here.

- Atomic propositions are written within quotes.
- F, G, U, X and W are written as capital letters `F`, `G`. `U`, `X` and `W`.
- $\vee$ is written as | (pipe character).
- $\wedge$ is written as `&`.
- $\Longrightarrow$ is written as `=>`.
- '(' and ')' can be used to group expressions.

[1] The repository of the cloning library can be found here: https://github.com/kostaskougios/cloning

[2] https://gitlab.lrz.de/i7/owl and https://owl.model.in.tum.de contain more information on recent research and development activities.

7.3.2. ATOMIC PROPOSITIONS IN OWL

In the previous section, we showed the Owl syntax. The Owl properties describe the behavior of a protocol module. Therefore all atomic propositions must contain information about possible communication behavior of the protocol implementation. We need to define a sender, receiver, message type, and communication direction (send or receive). To make more powerful LTL formulas, we also wanted to use wildcards for all characteristics. We created a parser for propositions that parses the following syntax: `"{role} {action} {messageType} {TO/FROM} {receiver}"`

- **`{role}`**. The name of the thread that initiates the communication action.
- **`{action}`**. `SEND` or `RECV`.
- **`{messageType}`**. The fully qualified name of the message type (e.g. `nl.this.packagename.Move`) or the short class name between '<' and '>' (e.g. `<Move>`). The short class name can only be used if the mapping to the fully qualified name is passed to the parser. More on this in Chapter 9.
- **`{TO/FROM}`**. `TO` or `FROM` keyword for readability.
- **`{receiver}`**. The name of the thread that should receive this message.

The wildcard operator '`*`' can be used on all positions. `{TO/FROM}` and `{receiver}` are optional (to support a more compact syntax). In addition to the above, the following shorthands can be used:

- **`"True"`**. All communication behavior is allowed to satisfy this property.
- **`"False"`**. No communication behavior is allowed.

LTL STATE MACHINES CREATED BY OWL

To create the final product state machine (done while model checking), we need to have a state machine representation of the LTL formula. The Owl parser can generate the LTL state machine for any given valid formula. This LTL state machine represents all runs of the protocol implementation that satisfy the formula. This state machine is stored as an LTS. This LTS consists of states and transitions. The transitions are labeled with propositions and acceptance identifiers. Observe that we chose to label transitions with propositions and not the states; more on this subject in Section 7.3.2. Those propositions describe possible communication behavior for that transition (also see Section 7.3.2).

To understand how LTL formulas are expanded to this LTS data structure, we will discuss the expansion of some base formulas. p and q are atomic propositions in the following examples.

EXPANSION OF 'p', 'p | q' AND 'p & b'

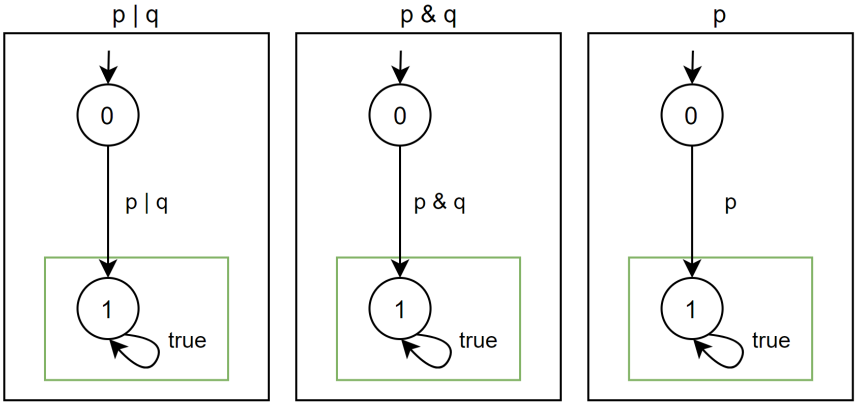

Figure 7.4: The LTL expansions for 'p', 'p | q' and 'p & b' as done by the Owl parser.

The expansion of 'p', 'p | q' and 'p & b' are relatively simple, see Figure 7.4. In all cases, the transition from the initial state must satisfy the formula's condition (transition from state 0 to state 1). After this transition, all actions are possible since the transition from state 1 to state 1 (loop) is labeled with 'true.' This transition (the loop from state 1) is part of the acceptance set. If a transition is part of an acceptance set, then the formula holds when that transition is found in a cycle of the product state machine (see Section 7.4.1 for its usage).

EXPANSION OF 'p U q', 'p W q' AND 'X p'

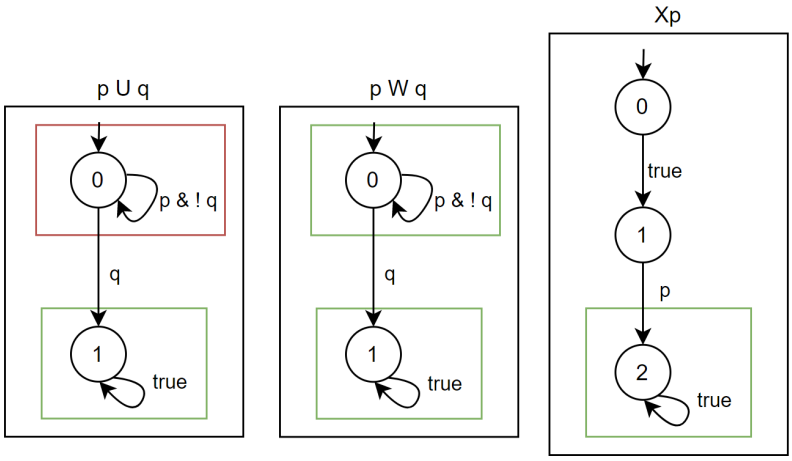

Figure 7.5: The LTL expansions for 'p U q', 'p W q' and 'X p as done by the Owl parser.

The expansions of 'a U b' and 'X p' are a bit more complex, see Figure 7.5. The expansion of 'X p' shows that a transition should be taken before the transition where p holds. The label 'true' on the transition between state 0 and 1 indicates that every action is possible to make this transition. The expansion of 'p U q' shows that p & !q must hold (loop transition from state 0 to state 0) until q holds once (transition from state 0 to state 1). After that, any actions are possible (loop from state 1 to state 1).

Other operators (e.g W or $\Longrightarrow$) can be written with only the operators discussed above, those equivalences are mentioned in Section 3.1. The expansion of W is given in Figure 7.5

to show the difference with the `U` operator. Expansions of more complex formulas are made by combining multiple expansions. Figure 7.6 shows an example expansion for the formula '$F(a|Xb)|Gc$' .

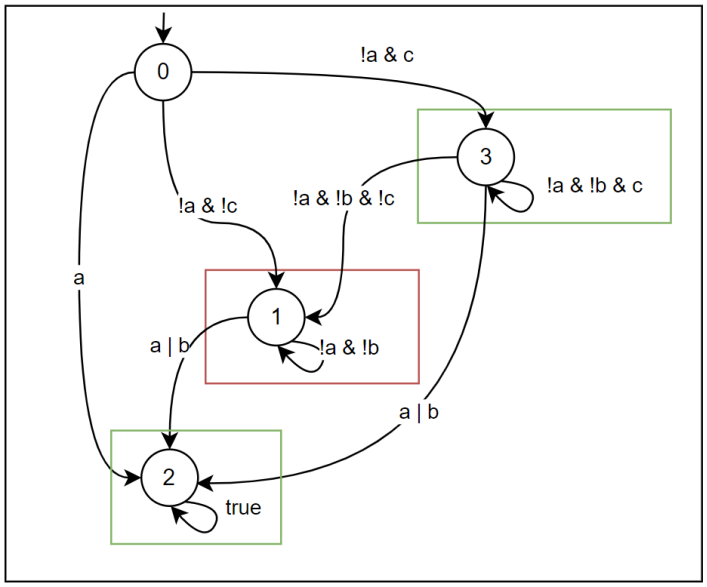

Figure 7.6: The LTL expansions for '$F(a|Xb)|Gc$' as done by the Owl parser.

LTL STATE MACHINE DATASTRUCTURE

We chose to explore the raw LTL state machine produced by the Owl library and map it to our own data structures to simplify the implementation of the model checking algorithm. Atomic propositions are parsed and also mapped to this data structure. The class diagram of this data structure is given in Figure 7.7. The `LtlTransitionExpression` objects are evaluated when determining possible communication actions during the execution of the model checking algorithm.

ON-THE-FLY CONSTRUCTION OF THE LTL STATE MACHINE

Note that we only construction the state-space in an on-the-fly manner. The LTL state machine is constructed in full before the execution of the algorithm. We also considered the on-the-fly construction of the LTL state machine, as shown in [15]. We chose not to do this for a couple of reasons:

- Implementation of such an algorithm is complex and time-consuming.
- It adds extra complexity to the model checking algorithm.
- We expect that the LTL state machines will generally not be very large compared to the protocol state machines.
- Testing with large LTL state machines is not the main aim of this research.

Owl. We chose to express formal properties in Owl, a DSL based on LTL. LTL is expressive enough for our requirements, and a parser for Owl to Java data structures is publicly available.

LABELING TRANSITIONS VS. LABELING STATES

We chose to model the LTL state machines as LTSs. So the states are only labeled with a number and not with propositions even though Owl supports both [31]. Because all possible communications define behavior in protocol implementations, it is most practical to label the transitions. The labels on the transitions resemble possible interactions.

Labeling. We chose to label transitions and not states in all state machines.

7.4. GUIDED DEPTH FIRST SEARCH

We only want to explore the part of the state-space that we need to check a specific property. This requires a more guided approach. This guided approach is the first part of the model checking algorithm.

Figure 7.8 shows a simplified version of the guided approach. The algorithm starts with passing a protocol copy and the LTL state machine's initial state to the `travelStateSpaceGuided` method. This method will explore the state-space guided by the LTL state machine. The steps the algorithm takes:

- Get all outgoing transitions from the current LTL state (Line 3).
- For every outgoing LTL transition: determine the set of possible exploring actions that satisfy the formula on the transition (Line 4).
- For every possible exploring action: check if it can be executed. (Line 11-20)
- If the action can be executed: check if this exploring action has been tried before in this context (Line 21). The `detectCycle` methods checks for the unique combination of protocol state, exploring action and LTL transition.
- When this exploring action has been tried before in this context, report a possible cycle and proceed to the nested cycle detection procedure (Line 23).
- If no cycle is detected, add this state to the `triedTransitionsStack` and call `travelStateSpaceGuided` recursively until we find a successor state in the product state machine or execution ends (Line 30-37).

Guided depth-first search. The product of the LTL state machine and the protocol state machine is determined by guided executions of the protocol implementation using the state-space explorer.

7.4.1. FINDING COUNTER EXAMPLES

The previous section described the detection of actions that have been executed before. When it is detected that a transition in the product state machine has been taken before, a subprocedure is called to check if it is part of a cycle. The starting state is marked before starting the nested search (passed as a parameter to the `travelStateSpaceGuidedForSecondCycle` method), see Figure 7.9. The state-space is explored again from the marked state. If the marked state is found when further exploring the state-space (in the same guided manner as discussed in Section 7.4), a cycle is found (Line 28). If one of the LTL transitions

in this cycle is accepting, then the formula holds for that trace (Line 31-35). This is a counterexample of the formula we are checking (remember that we were checking for traces of the negation of the formula we are checking).

The algorithm uses a stack to keep track of the trace it is currently exploring. When an accepting cycle is found, the algorithm returns false, and the trace stack is printed to the console.

Finding counterexamples. Nested cycle detection is used during the construction of the product state machine to identify accepting cycles. When found, this is logged to the console as the counterexample.

```
final CountDownLatch latch = new CountDownLatch(1);
AtomicReference<Optional> atomicReferenceValue = new AtomicReference<>(Optional.empty());

var baseThread = new Thread(() -> {
    synchronized (this.protocol){
        Thread thread = Thread.currentThread();
        new Thread(() -> {
            synchronized (this.protocol) {
                thread.interrupt();
            }
        }).start();
    }
    var self = this;

    try {
        // Action block
        if(actionToBeExecuted.direction == LtlTransitionExpressionAtomicPropositionDirection.SEND){
            self.environment.send(actionToBeExecuted.dummy, actionToBeExecuted.receiver);
            atomicReferenceValue.set(Optional.of(self.protocol));
            latch.countDown();
            return;
        }

        if(actionToBeExecuted.direction == LtlTransitionExpressionAtomicPropositionDirection.RECV){
            var result = self.environment.receive();
            if(result.getClass() == actionToBeExecuted.messageClass){
                atomicReferenceValue.set(Optional.of(self.protocol));
                latch.countDown();
                return;
            }
        }
        atomicReferenceValue.set(Optional.empty());
        latch.countDown();
        return;
        // End action block
    } catch (InterruptedException e) { // <<-- This means no possible action!
        Engine.LogTest("job was interrupted");
        atomicReferenceValue.set(Optional.empty());
        latch.countDown();
        return;
    } catch (Exception e2) {
        Engine.LogTest("caught other exception: " + e2.getCause());
        e2.printStackTrace();
        atomicReferenceValue.set(Optional.empty());
        latch.countDown();
        return;
    }
});

baseThread.start();

try {
    latch.await(); // Wait for countDown() in the thread.
} catch (InterruptedException e) {
    e.printStackTrace();
}

return atomicReferenceValue.get();
```

Figure 7.3: Detecting transitions

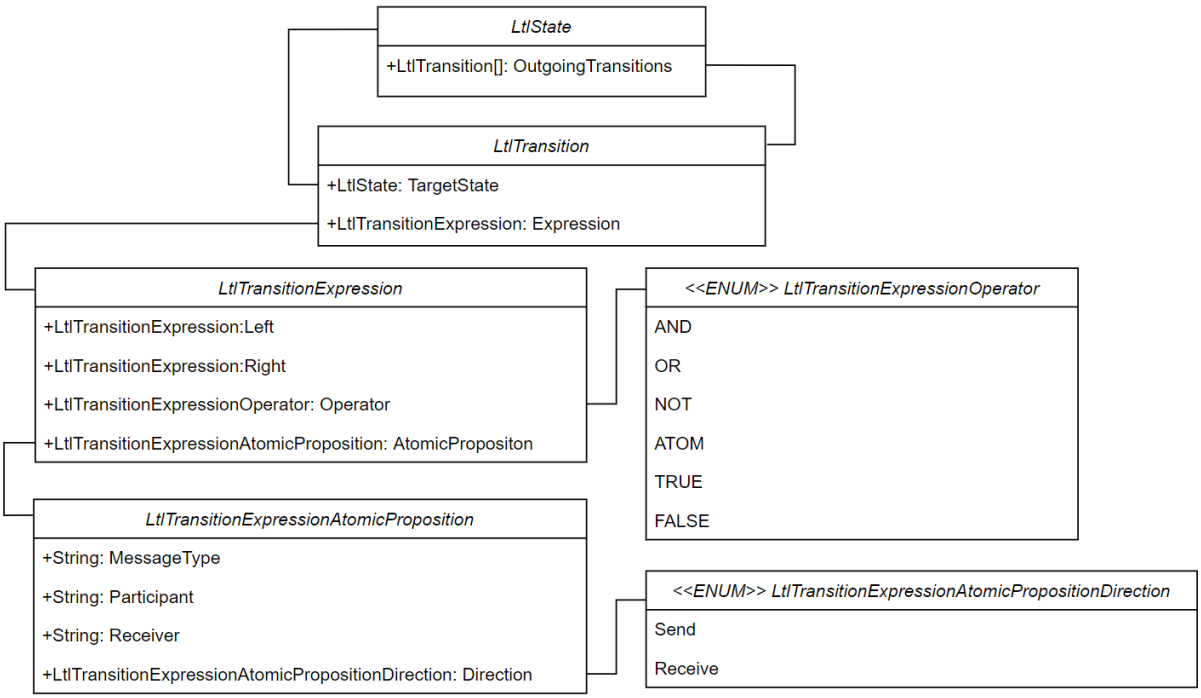

Figure 7.7: The LTL state machine data structure for Prut4j

```
1  private boolean travelStateSpaceGuided(Pr startingProtocolCopy, LtlState currentLtlState,
   ↪  Stack<TriedTransitionTuple> triedTransitionsStack) throws Exception {
2   // Try all outgoing transitions for the current ltl state
3   for (var transition : currentLtlState.OutgoingTransitions) {
4     for (var exploringAction :
     ↪  StateSpaceExploringActionsHelper.GetPossibleExploringActions(transition,this.exploringActions))
     ↪  {
5       // We have selected an action to explore (a possible transition on the protocol state machine)
6       // If no cycle is detected, we will try this action.
7       // Otherwise the exploration stops here, we found a cycle and we can stop exploring the graph
       ↪  for this sub trace.
8       var newTriedTransitionTuple = new TriedTransitionTuple(startingProtocolCopy.getState(),
       ↪  exploringAction, exploringAction.participant, transition);
9
10      // get the participating thread
11      var participatingThread = GetThreadForAction(exploringAction);
12
13      // set the protocol of the thread to a clone of the protocol
14      participatingThread.SetProtocolClone(StateSpaceExplorerHelper.deepClone(startingProtocolCopy));
15
16      // try to execute the action
17      Optional<Pr> optionalResultProtocol = participatingThread.ExecuteAction(exploringAction);
18
19      // if there is a result, we just took a transition in the protocol state machine
20      if (optionalResultProtocol.isPresent()) {
21        if (detectCycle(newTriedTransitionTuple,allTriedTransitions)) { // Detect possible cycle here
22          // We detected a transition that has been taken before.
23          var resultFromCycle2 = doCycleDetectionPhaseTwo(currentLtlState, transition,
           ↪  exploringAction, startingProtocolCopy, triedTransitionsStack);
24          // OR do second cycle directy to be able to report on the fly
25          if (resultFromCycle2) {
26            return true; // Only return true if a cycle is found, continue otherwise.
27          }
28        } else {
29          // we were able to make this transition, add to list to enable cycle detection
30          allTriedTransitions.add(newTriedTransitionTuple);
31
32          var protocolStateAfterTransition = optionalResultProtocol.get();
33
34          triedTransitionsStack.push(newTriedTransitionTuple);
35
36          // Call recursively to travel the whole protocol
37          var recursiveResult = travelStateSpaceGuided(protocolStateAfterTransition,
           ↪  transition.ltlTargetState, triedTransitionsStack);
38          if (recursiveResult) {
39       return true;
40          }
41        }
42      }
43    }
44   }
45   return false; // Returning false, we took all possible transitions and did not find any accepting
    ↪  cycles.
46 }
```

Figure 7.8: Create product state machine recursively

```
1  private boolean travelStateSpaceGuidedForSecondCycle(Pr startingProtocolCopy, LtlState
   ↪  currentLtlState, Set<TriedTransitionTuple> locallyTriedTransitions, TriedTransitionTuple
   ↪  markedTransitionTuple, Stack<TriedTransitionTuple> triedTransitionsStack) throws Exception {
2      //  Try all outgoing transitions for the current ltl state
3      for (var transition : currentLtlState.OutgoingTransitions) {
4          var possibleExploringActions =
           ↪  StateSpaceExploringActionsHelper.GetPossibleExploringActions(transition,this.exploringActions);
5          for (var exploringAction :  possibleExploringActions) {
6              //  We have selected an action to explore (a possible transition on the protocol state
               ↪  machine)
7              //  If no cycle is detected, we will try this action.
8              //  Otherwise the exploration stops here, we found a cycle and we can stop exploring the
               ↪  graph for this sub trace.
9              var newTriedTransitionTuple = new TriedTransitionTuple(startingProtocolCopy.getState(),
               ↪  exploringAction, exploringAction.participant, transition);
10
11             exploringAction.Print();
12
13             //  get the participating thread
14             var participatingThread = GetThreadForAction(exploringAction);
15
16             //  set the protocol of the thread to a clone of the protocol
17
               ↪  participatingThread.SetProtocolClone(StateSpaceExplorerHelper.deepClone(startingProtocolCopy));
18
19             //  try to execute the action
20             Optional<Pr> optionalResultProtocol = participatingThread.ExecuteAction(exploringAction);
21
22             //  if there is a result, we just took a transition in the protocol state machine
23             if (optionalResultProtocol.isPresent()) {
24
25                 var protocolStateAfterTransition = optionalResultProtocol.get();
26                 triedTransitionsStack.push(newTriedTransitionTuple);
27
28                 if (detectCycle(newTriedTransitionTuple, locallyTriedTransitions)) {
29                     if (newTriedTransitionTuple.equals(markedTransitionTuple)) {
30                         //  It is the state we are checking!
31                         var isAccepting  = isAccepting(triedTransitionsStack);
32                         if(isAccepting) {
33                             //  we found an accepting cycle
34                             return true;
35                         }
36                         else{
37                             //  Cycle is not accepting
38                             triedTransitionsStack.pop();
39                             return false;
40                         }
41                     } else {
42                         //  It is not the state we were checking for, we are ignoring this cycle!
43                         triedTransitionsStack.pop();
44                     }
45                 } else {
46                     locallyTriedTransitions.add(newTriedTransitionTuple);
47
48                     //  Call recursively to travel the whole protocol
49                     var recursiveResult = travelStateSpaceGuidedForSecondCycle(
50                             protocolStateAfterTransition,
51                             transition.ltlTargetState,
52                             locallyTriedTransitions,
53                             markedTransitionTuple,
54                             triedTransitionsStack);
55                     if (recursiveResult) {
56                         return true;
57                     }else{
58                         triedTransitionsStack.pop();
59                     }
60                 }
61             }
62         }
63     }
64     return false; // We took all possible transitions and did not find any accepting cycles.
65 }
```

Figure 7.9: Create product state machine recursively

8

CODE GENERATION

8.1. DESIGN

The example protocol implementation we created in Chapter 6 is relatively small. Creating those is already a laborious and time-consuming activity. We also argued that this manual task would become more complicated for a protocol with more interactions and participants. Therefore, we chose to generate the protocol implementations from a protocol definition in a DSL.

Being able to generate complex protocol implementations was also very useful when testing the model checking capabilities in the runtime. Otherwise, it would only be tested with our manually created implementations.

ABSTRACT SYNTAX TREE DATA STRUCTURE

We chose to use an abstract syntax tree (AST) as an intermediate data structure for code generation. Our approach is very similar to the tree-walk[1] approach in ANTLR, a well-known parser and code generator library. The AST we defined represents the protocol implementation on a course-grained level. The class diagram of the AST classes is given in Figure 8.1. Every class corresponds with a specific part of the protocol implementation class we discussed in Chapter 6 and contains all the information needed to generate that specific part of the code. For example, the `ASTProtocol` class has a `String` field for the name of the protocol to create the code defining the class and has all child objects necessary to create the code of the full class.

We step through the whole AST to generate a protocol implementation class. We want to support multiple target languages, revisions, or alternatives in the target syntax. Therefore, the generation logic for an AST object is not defined in the class itself but in adapters. For Prut4j, we created adapters for both the strict and the liberal protocol implementations. Aside from the flexibility of using ASTs, there are also other possible advantages of ASTs. For example, ASTs can be useful for static analysis. We could use it to obtain advanced metrics (e.g. [33]). Such applications might be a subject for further research but is beyond the scope of this report.

Abstract syntax trees. We use an intermediate AST data structure to support multiple configurations or languages.

[1] See `https://www.antlr2.org/doc/sor.html#_bb2` for more information about the tree-walk approach.

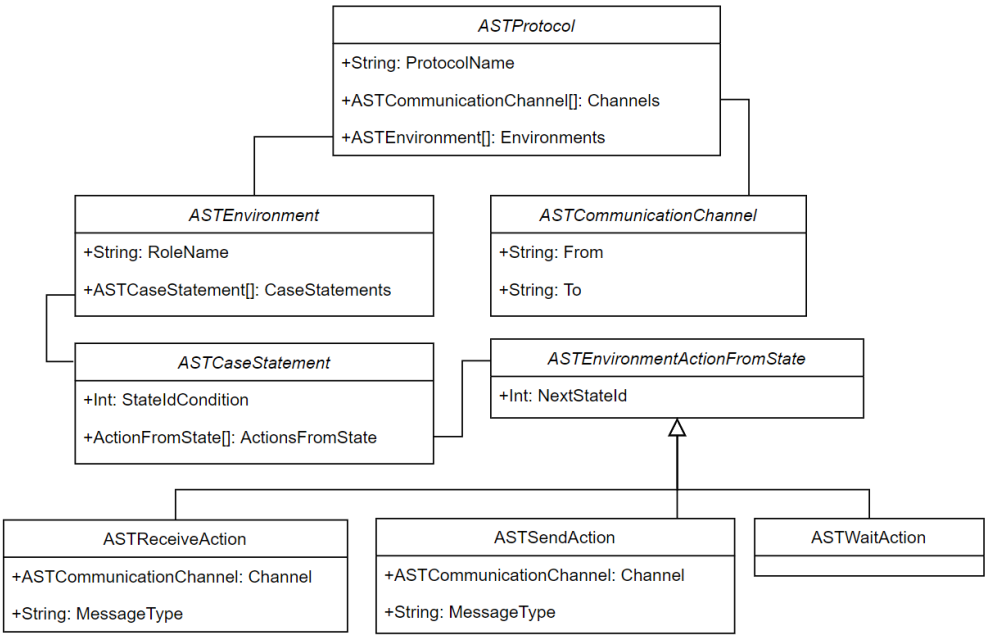

Figure 8.1: Abstract Syntax Tree Data Structure Class Diagram

8.1.1. DISCOURJE, DSL FOR PROTOCOL DEFINITIONS

As we mentioned in Section 2.3, we do not want to create the protocol implementation manually but generate them from a DSL like Reo or Scribble. Since creating a DSL and creating a parser for the DSL is not the focus of our research, we used an existing parser for protocol definitions. We chose to use Discourje [19] for a couple of reasons:

- **Compatibility with Java.** Discourje has (with some minor changes) the ability to parse a definition to a Java data structure, making it practically useful for further implementation.
- **Global state machine generation.** The parser for Discourje outputs an LTS. This LTS represents the global state machine for the protocol. The most important classes of this data structure are State and Transitions. The State class has a numerical state identifier and a list of outgoing Transitions. The Transition class has a reference to a target state and metadata that describes the communication behavior, like the type of the message and the receiver. This is all the information we need to generate a protocol implementation.
- **Existing protocol definitions.** Protocol specifications for publicly available benchmark programs, network topologies, and some well-known games are available in the Discourje project. This was of significant importance for the validation of our proof of concept. This will be discussed in-depth in Chapters 10 and 11.
- **Feature-rich.** Discourje is feature-rich. It supports many constructs like role indexing (parameterization) and repetitions (e.g., loops).
- **Scientifically backed.** Discourje has been created by researchers and their research shows promising results regarding its applicability in protocol programming.

An example protocol definition in Discourje has already been given in Figure 5.2.

DSL for protocol programming. We selected Discourje as our DSL for protocol definitions. The Discourje parser and protocol definitions for various applications are publicly available.

8.2. IMPLEMENTATION

CREATING THE ABSTRACT SYNTAX TREE

To start building the AST, we needed a way to obtain information from the protocol state machine. First, we need the coarse-grained information from the protocol state machine, like a list of all the participants, so we know what environments we need to generate later. Then we need to know all possible communication between participants so we can generate the communication channels. Once we have that information, we can start building the environment for every participant.

To obtain all this information, we need a way to search the whole graph data structure. We chose to use a visitor pattern. When visiting all nodes in the graph, we gather all the information we need to build the protocol's AST. We chose to do two depth-first searches (two full visits of the protocol state machine). This made the implementation more straightforward. If we used a single depth-first search, we would need to add environments and channels as we encounter them. This will over-complicate the code.

THE DEPTH-FIRST SEARCH

To explore all `ProtocolStates` and `Transitions` while obtaining various pieces of information, we implemented the Visitor pattern extended with simple cycle detection. To this end, we implemented a visitor pattern that accepts multiple visitors. A visitor must implement the `visit(state)` method, where it can process the information of the current `ProtocolState`.

THE FIRST PASS

We use the first pass to obtain some general info from the protocol definition. We created a few visitors to obtain this information. In the first place, we extracted the roles in the protocol. Secondly, we extracted all possible communication channels between roles by checking all transitions in the LTS. With the information we gathered in this pass, we can create some high-level AST items like the `ASTCommunicationChannels`, but we need more detailed information about the actual communication behavior. We will obtain this with a second pass.

THE SECOND PASS

In the second pass, we obtain the detailed information we need to create a full AST for the protocol. In contrast to the first pass, we now create a visitor for each participant. Every visitor will be able to create the `ASTEnvironment` object for that participant after a full visit of the protocol. Every visitor gets a set of available communication channels (in and out) for the specific environment. We already obtained the information of those channels in the first search.

The environment will create an `ASTCaseStatement` for every visited state. Then the environment visitor needs to determine for each protocol state what actions are possible from the state's environment. This could be a send or receive action and a wait action if there is

no communication for that role in that state. The environment is not supposed to communicate at that moment. Once determined what actions are possible, an `ASTEnvironmentActionFromState` is added for every action. When the second pass is done, the final `ASTProtocol` object is created with the result from all visitors.

Creating an AST. The Discourje parser parses protocol definitions to an LTS. An AST data structure is extracted using full visits of the LTS.

GENERATE CODE FROM THE ABSTRACT SYNTAX TREE

Once we have the full AST, we can start generating code. All AST classes implement a base class with a `buildSyntax()` method. That method builds the syntax of that specific syntax tree item. The actual implementation of the `buildSyntax()` method is delegated to language or configuration-specific adapters to allow support for multiple target languages or configurations (See Section 8.2 for an example usage). Some adapters will need to call the `buildSyntax()` method of certain child items. We implemented adapters for Java for every syntax tree item in the data structure. When building support for another object-oriented programming languages like C#, one would need to create a separate writer for each syntax tree item, and we would need to find equivalents for all behavior in the protocol implementation.

An example writer class is given in Figure 8.2. The `buildSyntax()` writes the Java code for the root of the syntax three, the `ASTProtocol`. It will generate the protocol class code (Line 4 -16) and will call the `buildSyntax()` method of child objects to generate the code of the message queues and the environments (Line 24). The writer for the `ASTEnvironments` will generate the switch statement and call the `buildSyntax()` method for its child objects.

The AST items are relatively coarse-grained. We chose to keep it coarse-grained to keep the detailed implementation in the adapters for the specific programming languages or revisions.

CODE WRITER ADAPTERS FOR LIBERAL PROTOCOL IMPLEMENTATIONS

We created different adapters for creating strict and liberal protocol implementation. The strict implementation uses a single state machine for all participants, so it does no post-processing on the AST before generating the protocol class. The second decomposition-based liberal implementation first generates a local state machine for every participant (see Section 6.2.3).

Generating a Java class. The AST is used to generate a protocol implementation Java class. We use different writers for the strict and optimized liberal implementations. The liberal version uses an algorithm to determine the local projection of the global state machine for every participant.

```
1  public class ProtocolWriterForJava11 implements ISyntaxWriter<ASTProtocol> {
2      @Override
3      public void buildSyntax(StringBuilder builder, int tabCount, ASTProtocol SyntaxTreeItem) {
4        StringBuilderSyntaxHelper.addLine(builder, tabCount,"/* !!! IMPORTANT !!!");
5        StringBuilderSyntaxHelper.addLine(builder, tabCount," * !!! This code is generated from a
         ↪  protocol definition. !!!");
6        StringBuilderSyntaxHelper.addLine(builder, tabCount," * !!! Any Changes made to this code could
         ↪  be overridden. !!!");
7        StringBuilderSyntaxHelper.addLine(builder, tabCount," * !!! If you want to change the protocol,
         ↪  change its definition and regenerate this code. !!!");
8        StringBuilderSyntaxHelper.addLine(builder, tabCount," **/");
9
10       StringBuilderSyntaxHelper.addLine(builder, tabCount,"package
         ↪  nl.florianslob.modelchecking.generated;");
11
12       ... // imports etc. ommitted
13
14       StringBuilderSyntaxHelper.addEmptyLine(builder, tabCount);
15
16       StringBuilderSyntaxHelperForJava11.addScopedBlock(builder,"public class
         ↪  "+SyntaxTreeItem.protocolName+" implements Pr ", tabCount,
17         (tabCountLvl0) -> {
18           //  Order all Environments by name, this is not necessary, but makes the generated code more
             ↪  readable and consistent.
19           for(ASTEnvironment ASTEnvironment : SyntaxTreeItem.environments){
20             StringBuilderSyntaxHelper.addLine(builder,tabCountLvl0, "private final
               ↪  BlockingQueue<ProtocolMessage> "+ASTEnvironment.roleName+"Queue = new
               ↪  LinkedBlockingQueue<>();");
21           }
22
23           //  Build the syntax of each environment
24           for(ASTEnvironment ASTEnvironment : SyntaxTreeItem.environments){
25             ASTEnvironment.buildSyntax(builder,tabCountLvl0);
26           }
27
28           StringBuilderSyntaxHelper.addEmptyLine(builder, tabCountLvl0);
29
30           StringBuilderSyntaxHelperForJava11.addMethodOverride(builder,"public IEnvironment
             ↪  getEnvironment(String environmentName) throws Exception", tabCountLvl0,
31             (tabCountLvl1) -> {
32               StringBuilderSyntaxHelperForJava11.addScopedBlock(builder,"switch (environmentName)",
                 ↪  tabCountLvl1,
33                 (tabCountLvl2) -> {
34                   for(ASTEnvironment ASTEnvironment : SyntaxTreeItem.environments){
35                     StringBuilderSyntaxHelper.addLine(builder,tabCountLvl2, "case
                       ↪  "+ASTEnvironment.roleName+": return "+ASTEnvironment.roleName+"Environment;");
36                   }
37
38                   //  Add default to switch case statement.
39                   StringBuilderSyntaxHelper.addLine(builder, tabCountLvl2, "default: throw new
                     ↪  Exception(Ünknown environment);");
40                 }
41               );
42             }
43           );
44
45           StringBuilderSyntaxHelperForJava11.addMethodOverride(builder,"public String[]
             ↪  threadNames()", tabCountLvl0,
46             (tabCountLvl1) -> {
47               ...
48             }
49           );
50
51           ...
52         }
53       );
54     }
55 }
```

Figure 8.2: Code generation in a Java code writer for the `ASTProtocol` class

9

PUTTING IT ALL TOGETHER

In this chapter, we summarize all tools that are developed [1].

- **Owl** This project contains a copy of the Owl Library with some small changes to make it compatible with Prut4j.
- **Discourje** This project contains a copy of the Discourje project with some changes to make it compatible with Prut4j. This tool is written in Clojure and builds to a .jar file. This file is imported as an external library in the CodeGenerator project.
- **Prut4j.API** This library project contains all shared interfaces and classes. The most important one is the `Pr` interface. This is the only library that target applications need to reference. This project has no external dependencies to have maximal compatibility.
- **Prut4j.Runtime** This project contains the `Engine` class that can check Owl properties on protocols that implement the `Pr` interface.
- **Prut4j.CodeGenerator** Uses the Discourje library to parse `.dcj` files and generates protocol implementations into the Prut4j.Tests and Prut4j.Benchmarks projects.
- **Prut4j.Tests** The code generator generates some protocol implementations into this project. This project contains Owl properties for all those protocol implementations, see Chapter 10.
- **Prut4j.Benchmarks** This project contains two copies of a well-known benchmarking application. One copy is untouched and acts as a reference implementation. The other copy uses protocol implementations generated by the code generator. See Chapter 11.

Proof of concept implementation summarized. Prut4j consists of a parser for Discourje, a parser for Owl, a code generator, a runtime library, and an API library.

[1] Sources are publicly available at https://github.com/FlorianSlob/Prut4j

MODERN BUILD PIPELINES

The Prut4j repository is hosted in Github and has a build pipeline in Azure DevOps Pipelines[2] (running for every pull request to the master branch). Running the Owl unit tests in the Prut4j.Tests project is as easy as running regular JUnit tests. An example build configuration to run the test in the test project in given in Figure 9.1.

```
1   trigger:
2   - master
3
4   pool:
5     vmImage: `ubuntu-latest'
6
7   steps:
8   - task: Gradle@2
9     inputs:
10      workingDirectory: `tests' # Directory of the test project
11      gradleWrapperFile: `tests/gradlew'
12      gradleOptions: `-Xmx3072m'
13      javaHomeOption: `JDKVersion'
14      jdkVersionOption: `1.12'
15      jdkArchitectureOption: `x64'
16      publishJUnitResults: false
17      testResultsFiles: `**/TEST-*.xml'
18      tasks: `build'
```

Figure 9.1: Example build configuration for Azure Devops

The Prut4j build pipeline is running on the free tier of Azure DevOps, so we are limited in the 'build minutes' we can use. Therefore we excluded the Prut4j tests project from the build pipeline. Enterprise environments will be less limited. The build that contains those long-running unit tests can also be run only on a specific schedule or on-demand.

[2] See https://azure.microsoft.com/en-us/services/devops/pipelines/ for more information about Azure DevOps Pipelines

III

CONTRIBUTION - VALIDATION AND CONCLUSION

10

EXPRESSIVENESS AND APPLICABILITY

To validate the expressiveness of Prut4j, we did some case studies. We used those case studies to investigate several different aspects. In the first case study, we studied six common network topologies to investigate the distinguishing power of model checking. We generated a protocol implementation for every topology and wrote a set of LTL properties where every topology passes a unique combination of those properties. In the second case study, we studied the ability to test complex flows in the protocols. To this aim, we generated protocol implementations for several games and wrote properties to check some generic (e.g., boundedness and liveness) and some game-specific properties. Our last case study aimed to test the ability to use Prut4j in a real, existing program. We used an existing third-party benchmark suite and replaced existing communication behavior with calls to a generated protocol implementation. In total, we created fourteen protocol modules in Discourje and wrote 148 unit tests in Owl.

10.1. CASE STUDY: NETWORK TOPOLOGIES

This case study aims to analyze the *distinguishing power* of LTL properties written in Owl. In other words, we want to show that an LTL expression in Owl can hold for a specific protocol module and be violated by others. We created six protocol modules for common network topologies. The number of threads (denoted as n) is parametrized for those protocol modules. Because every network is unique for n=4, we chose to use four threads. The network topologies we implemented are Directed ring, Undirected ring, Star, Binary tree, Full mesh and 2D mesh. Protocol definitions in Discourje for every protocol module are given in the supplemental materials, see Appendix A.

For every protocol module, we checked the following Owl properties:

1.
```
G("worker_1_ RECV <Boolean>" =>
    X("worker_1_ SEND <Boolean> TO worker_2_"))
```

2.
```
G("worker_1_ RECV <Boolean>" =>
    X("worker_1_ SEND <Boolean> TO worker_0_" ||
        "worker_1_ SEND <Boolean> TO worker_2_"))
```

3.
```
G("worker_1_ RECV <Boolean>" =>
    X("worker_1_ SEND <Boolean> TO worker_0_"))
```

4.
```
(G("worker_1_ RECV <Boolean>" =>
    X("worker_1_ SEND <Boolean> TO worker_0_" ||
        "worker_1_ SEND <Boolean> TO worker_3_")))
&&
(G("worker_2_ RECV <Boolean>" =>
    X("worker_2_ SEND <Boolean> TO worker_0_")))
```

5.
```
G("worker_1_ RECV <Boolean>" =>
    X("worker_1_ SEND <Boolean> TO worker_0_" ||
        "worker_1_ SEND <Boolean> TO worker_2_" ||
        "worker_1_ SEND <Boolean> TO worker_3_"))
```

6.
```
(G("worker_1_ RECV <Boolean>" =>
    X("worker_1_ SEND <Boolean> TO worker_0_" ||
        "worker_1_ SEND <Boolean> TO worker_3_")))
&&
(G("worker_2_ RECV <Boolean>" =>
    X("worker_2_ SEND <Boolean> TO worker_0_" ||
        "worker_2_ SEND <Boolean> TO worker_3_")))
```

Property	Directed ring	Undirected ring	Star	Binary tree	Full mesh	2D mesh
1	✔	🐞	🐞	🐞	🐞	🐞
2	✔	✔	✔	🐞	🐞	🐞
3	🐞	🐞	✔	🐞	🐞	🐞
4	🐞	🐞	✔	✔	🐞	🐞
5	✔	✔	✔	✔	✔	✔
6	🐞	🐞	✔	✔	🐞	✔

Table 10.1: Test results for different network topologies

The test results in Table 10.1 show the results that we expected. Every protocol module has a unique set of properties that hold.

Distinguishing power. The network topologies case study shows the distinguishing power of Owl unit tests in Prut4j.

10.2. CASE STUDY: GAMES

We created and tested three protocol modules to show that it is possible to create and test protocol modules with complex control flows in Prut4j. We implemented some protocol modules in Discourje for some well-known games. We were able to define and check various Owl properties. Some properties were generic properties that show aspects like the boundedness of channels/ queues, that every sent message is eventually received and liveness. Other validated Owl properties prove some specific characteristics of a protocol module. We will discuss some of those in the next sections.

10.2.1. TURN-TAKING

This is the protocol with the lowest complexity and has a fixed number of participating threads. This is the protocol module we already used to implement Chess, but it can also be used for games like Tic-Tac-Toe. One property we are checking is the following:

```
G("black SEND <Move> TO white" =>
    X((!"black SEND <Move> TO white") U "white SEND <Move> TO black"))
```

Here we validate that when Black sends a message, it will not send a message again until White has sent a message.

10.2.2. ROCK-PAPER-SCISSORS

The number of participating threads is parameterized for this protocol. The number of active threads decreases over time (e.g., when a thread loses a round). For this protocol, we were able to check some specific properties. One example is checking that the loser of a round can drop out of the game. Another example:

```
(!"player_0_ RECV *" U "player_0_ SEND * TO player_1_") &&
    (!"player_0_ RECV *" U "player_0_ SEND * TO player_2_")
```

This property states that player_0_ will not receive a message until it has sent a message to the other players.

10.2.3. GO-FISH

The number of participating threads is parameterized for this protocol. Threads do not take turns in a statically fixed order. This is dynamically determined while playing. For this protocol implementation, we were also able to test several specific properties. One example is the rule that if a thread does not have the asked card, it starts with the next turn. Another example is eventual reception:

```
G("player_0_ SEND * TO player_1_" => X(F("player_1_ RECV *")))
```

This property states that if player_0_ sends a message, player_1_ will eventually receive a message. All protocol definitions in Discourje and the tests in Owl files are available in the supplemental materials, see Appendix A.

Create and test complex flows. The game case studies show that Prut4j can be used to create and validate protocols with complex flows.

10.3. CASE STUDY: SCIENTIFIC KERNELS

We replaced the communication behavior in four scientific kernels (in computational fluid dynamics) with Prut4j protocol modules to show Prut4j's ability to integrate into existing real-world programs easily. The four scientific kernels are part of the Numerical Aerodynamic Simulation Parallel benchmarks (NPB) [14]. The NPB is a benchmark suite that was designed to test the performance of parallel compute architectures. All protocol definitions are parameterized for a number of threads, denoted with n. We cite a brief explanation of the benchmarking programs [14]:

- **'CG** uses a **C**onjugate **G**radient method to compute approximation to the smallest eigenvalues of a sparse unstructured matrix. This kernel tests unstructured computations and communication'
- **'FT** contains the computational kernel of a 3-D Fast **F**ourier **T**ransform (FFT). Each FT iteration performs three sets of one-dimensional FFTs, on sets for each dimension.'
- **'IS** performs sorting of integer keys using a linear-time **I**nteger **S**orting algorithm based on computation of the key histogram.'
- **'MG** uses a V-cycle Multi Grid method to compute the solution of the 3-d scalar Poisson equation. The algorithm works iteratively on a set of grids that are made between the coarsest and finest grids. It tests both short and long distance data movement. '

All benchmark implementations communicate among threads using messages over shared memory. The order and the type of those messages are important. This is what makes those implementations useful for our research. All implementations have a master thread where communication starts. The CG and IS implementation send and receive variables to and from one or more worker threads until calculations are complete. The FT and MG benchmark implementations orchestrate multiple specialized workers (two and four, respectively). Those specialized workers execute different calculations and have a specific role in the protocol. The exact calculations that those workers execute are not relevant to our research. We only show that it is possible to replace this existing communication behavior with calls to a `Pr` object. To update the implementations, we did the following:

- **Generate protocol modules from the Discourje definitions.** The Discourje definitions were already available in the Discourje project and can be found in the supplemental materials, see Appendix A.
- **Test protocol modules.** We wrote a set of LTL properties for every protocol module to test different aspects, similar to those made to test the games' protocol modules. The Owl files for those properties can be found in the supplemental materials, see Appendix A. We had to think about the protocols while writing the Owl properties for those protocol modules. It is interesting to note that while doing so, we found bugs in the existing Discourje code, even before checking the LTL properties.
- **Replace protocol code and update startup procedure.** Remove all the protocol code that is intertwined in all separate threads and replaced them with calls to a `Pr` protocol object variable. Update the startup procedure of the benchmark implementation to set the protocol object variable to the required generated Prut4j protocol module (an implementation of the `Pr` interface).

Channel or queue-based protocols for scientific kernels have not been validated before, as far as we know. Those scientific kernels are often concurrent programs, and testing the protocols in those kernels is therefore very relevant. There are examples of publications in well-known journals like Science and PNAS that have been retracted because of bugs [36].

Applicability. The NPB implementation case study shows that it is possible to replace existing communication logic with a generated and validated protocol implementation.

11

PERFORMANCE

11.1. TEST EXECUTION

Brand	Type	Processor	Clock	Cores	Ram	Mean execution time
Xiaomi	TM1703	Intel Core i7-8550U	1.80GHz	4	8Gb	47s
Dell	XPS 15 9570	Intel Core i9-8950HK	2.90Ghz	6	32Gb	45s

Table 11.1: Development laptop specification and the mean execution time.

The underlying state machines of generated protocol implementations can grow exponentially when using a larger number of threads in parameterized protocol definitions. This means that a possible state-space explosion can occur while checking the Owl properties. Validating those protocol implementations could take a significant amount of time. We deem it realistic to use a low number of threads when validating the protocol implementations. We assume that if it works for two to four threads, then it will work for a higher number of threads. With a low number of threads (between two and four), execution times are within reasonable bounds. We measured the mean execution time of all 148 unit tests we created during our case studies on two available developer laptops. Table 11.1 outlines the specifications and the mean execution times for both machines. Test summaries are available in the supplemental materials, see Appendix A. Even though this is slow compared to conventional unit tests, we think that this is acceptable since:

- Protocols are complex functionality.
- Testing with Prut4j provides maximal coverage.
- The tests are executable on average developer machines.
- Testing the protocols manually is less feasible.
- There is still room for optimizations (e.g., caching), see Chapter 13.
- Execution in modern build pipelines like Azure DevOps Pipelines is still possible.

Model checking algorithm execution performance. Unit-tests execution is fast enough to perform on the machine of a developer. The state-space explosion is still an issue when complexity increases or the number of participants grows. Further optimizations are possible.

11.2. RUN-TIME OVERHEAD

Aside from the performance of running tests, we also wanted to validate the performance during real executions. Real execution is likely to use more threads than used during unit testing. To this aim, we used the publicly available Java implementation of the NPB as a reference implementation to compare the performance of the generated protocol implementations [14]. The NPB was created to evaluate the performance of parallel systems using shared-memory multiprocessors. The Java NPB consists of seven different benchmarks, all testing performance by completing a set of computational tasks. The benchmarks can be run on different systems, and one can compare the result to compare those systems' performances.

We already adapted four benchmark programs to use Prut4j protocol implementations for the case study discussed in Chapter 10.

EXECUTION ON SUPERCOMPUTER

To check run-time performance, we executed the original and adapted implementations for a varying number of threads on a node of the Dutch national supercomputer. The specification of this node can be found in Table 11.2[1]. We measured slowdowns and speedups relative to the reference implementations. We found that the strict protocol implementations perform poorly when using a high number of threads (up to 4× slowdown in the worst case). However, the liberal decomposition-based protocol implementations show little performance decrease (below 5 %) and, in some cases, even a performance increase. We did not expect to see a performance increase, and it is therefore worth further investigation.

Node Type	Cores	CPU	Clock	Memory in Gb	SBU / node-hr
thin	24	E5-2690 v3	2.6 GHz	64	24

Table 11.2: Specification of the Cartesius supercomputer node.

EXECUTION RESULTS FOR STRICT AND LIBERAL PROTOCOL IMPLEMENTATIONS

To compare the performance of the reference implementation and the updated implementation on the supercomputer, all implementations are run thirty times for a different number of threads (2 to 24 for all even numbers). Figures 11.1, 11.2, 11.3 and 11.4 show slowdowns (if $y>1$) and speedups (if $y<1$) of the Prut4j-based implementations (strict and liberal) relative to the reference programs (computed using mean execution-times). The raw result data is available in the supplemental materials, see Appendix A. This also contains the means and standard deviations. Standard deviations do not affect the trends since they are mostly below 5% of the mean.

We discuss three main observations:

[1] See https://userinfo.surfsara.nl/systems/cartesius/description for more information about Cartesius.

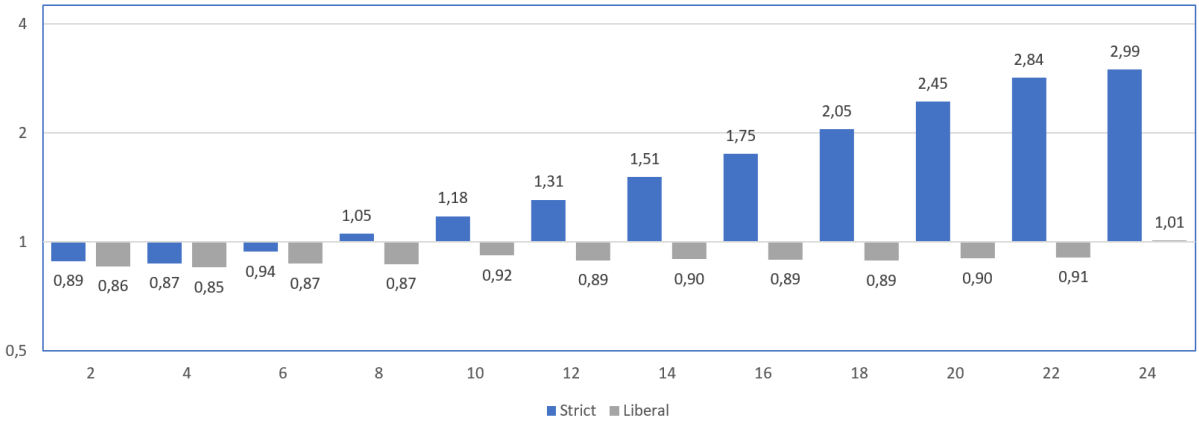

Figure 11.1: Slowdown (y) of Prut4j-based versions of kernels CG relative to reference programs, against numbers of threads (x)

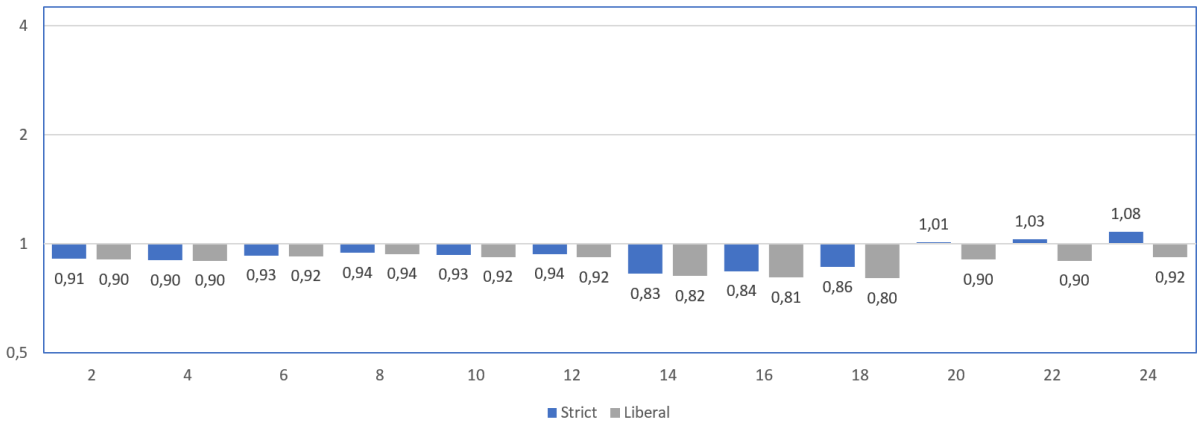

Figure 11.2: Slowdown (y) of Prut4j-based versions of kernels FT relative to reference programs, against numbers of threads (x)

- **Strict protocol implementation slow-downs are problematic.** Execution-times increase significantly for the kernels CG (Figure 11.1) and MG (Figure 11.4) for a higher number of threads. A slow-down of almost 4× is observed in the worst case (24 threads in the MG implementations, see Figure 11.4). The CG and MG programs have more communication among threads compared to FT and IS and are therefore more susceptible to slow-downs due to the extra abstraction layer in the protocol implementation.

- **The optimizations with the liberal protocol implementations are very effective.** We observe no increasing curve for CG an MG, and the maximum slow-down is no more than 1.05×.

- **In many cases, we observe speedups.** In many cases, the programs with the generated liberal protocol implementations are faster than the hand-written reference program, particularly for IS. We did not expect this observation and calls for further research. This optimization could be because of the liberal nature of the protocol implementations. Threads can do some actions directly, where they would otherwise have to wait when using the strict protocol implementations or hand-written communication structures. Another possible explanation could be that the memory

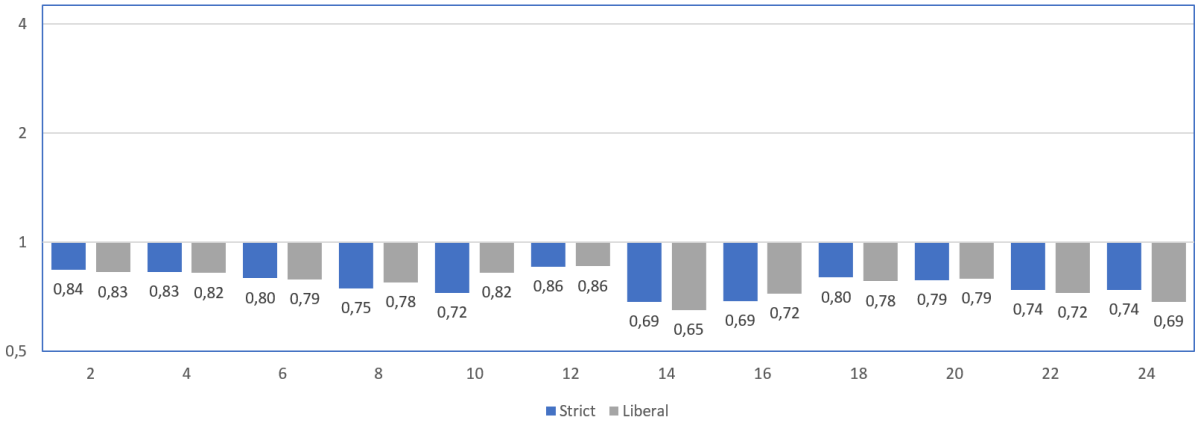

Figure 11.3: Slowdown (y) of Prut4j-based versions of kernels IS relative to reference programs, against numbers of threads (x)

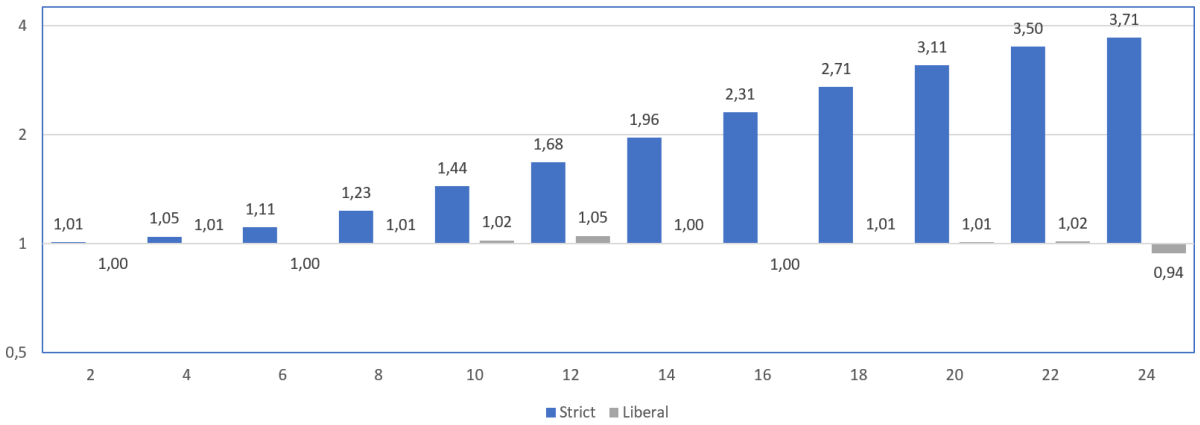

Figure 11.4: Slowdown (y) of Prut4j-based versions of kernels MG relative to reference programs, against numbers of threads (x)

allocation behavior is changed so that it is handled differently by the Java garbage collector.

Performance on the multi-core system. The base implementation performs poorly with a high number of threads (24) for some benchmarking programs. The optimized implementation had a small performance loss, below 5%. Some programs even gain performance.

12

Conclusion

It is time to reflect on our research question: *How can adaptation-based model checking be applied in the context of a domain-specific language for protocols?*

We found that in order to model check protocol implementations, we needed to express formulas in some formal logic. We selected the DSL Owl to formulate LTL properties that can describe the communication behavior of protocol implementations. To check those LTL properties, we created an Engine to execute an adaptation-based model checking algorithm.

The model checking algorithm explores the state-space in an adaptation-based manner using real executions of the protocol implementations. Those protocol implementations can be generated from a protocol definition in Discourje. Those higher-level code generation and compilation techniques do not influence the outcome of the model checking algorithm because of this adaptation-based approach.

The Owl parser, Code generator (based on Discourje), and the model checking engine are part of the research prototype tool Prut4j. With Prut4j we were able to generate and test complex protocol implementations that would otherwise be very laborious to do manually.

The results of the case studies indicate that the expressiveness and applicability of Prut4j are promising. We can write LTL properties for protocol implementations, check those with maximal coverage, and identify a protocol's distinctive behavior. Writing tests by hand with similar coverage would have been practically impossible. Results show that we can create and test protocols with complex communication flow and easily integrate a generated and validated protocol implementation into real-world existing programs. We have also shown that the protocol implementations generated by Prut4j show acceptable performance during run-time compared to third-party reference implementations, especially for the liberal protocol implementations.

We believe that Prut4j, even with its current limitations, shows that an adaptation-based approach to model checking protocol implementations in the context of domain-specific languages is feasible and can be very powerful. Therefore, we encourage further development of tools and research on this subject.

13

FUTURE WORK

We concluded that we deem the adaptation-based approach to model checking protocol implementations feasible and encourage further research and development. In this chapter, we will outline some possible new research subjects or developments.

PRODUCTION-READY

Prut4j has been developed as a research prototype to show the possibilities of an adaptation-based approach to model checking. We specifically aimed to show its expressiveness and performance in real-world applications. This means that we took some short-cuts when it comes to usability from a developer's perspective. This could be improved to make Prut4j production-ready.

DSL FEATURES

Discourje and Owl proved to be a good choice for our research. However, it might not be the best choice from a developer's perspective. Possible improvements could be better usability and integration, simpler syntax, less boilerplate code or improved DSL features.

LANGUAGE SUPPORT

In Chapter 8, where we discussed the code generation algorithm, we already talked briefly about support for different programming languages and configurations. While we argued that the code generation algorithm could be extended with additional languages, this is still something that needs to be validated. When it comes to code generation, equivalent logic could be created for languages like C#, Go, or Python. Syntax shall be different, and some other concepts like the `BlockingQueue` in Java will need to be replaced with an equivalent in that language. Aside from code generation, much research could be done regarding the Engine. The Engine is now language-specific. When adding support for additional languages, one needs to research ways to explore the state machine of the protocol implementation for that target language.

WARNINGS FOR LIBERAL PROTOCOL IMPLEMENTATIONS

In Chapter 6, we discussed the strict and liberal versions of the protocol implementations. We argued that the decomposition-based liberal protocol implementations could lose semantics compared to the strict protocol implementations. It might be possible to extend the current algorithm to detect when semantics are lost. Developers could be warned when using an optimization that is not semantics preserving.

OPTIMIZATIONS TO THE MODEL CHECKING ENGINE

In Chapter 11, we discussed the performance of the model checking engine (the running time of the algorithm). We deemed the running times acceptable, but we think there is also room for optimizations.

One possible optimization is caching. For every protocol implementation, we check multiple properties. We reconstruct the protocol implementation state machine from scratch for every execution of the algorithm. This is not ideal, so there is possible profit when implementing caching techniques.

Another possible optimization is the use of hashing. In the current implementation, visited transitions (combinations of a protocol copy, LTL transition, and an exploring action) are saved in a set of Tuples. With the low number of threads we are using, this did not cause any memory issues. However, the set of visited states could grow exponentially when adding extra threads. Memory use could become an issue. Hashing techniques could lower memory usage by only saving a hash of the visited transition to memory. One must keep in mind that hash collisions are possible and could cause spurious outcomes.

EXPRESSIVENESS OF LTL

In Chapter 7, we discussed our choice for LTL as a formal language to express temporal properties. We understand that other formal languages could be more expressive and, therefore, possibly more valuable. Other formal languages require other model checking algorithms. The state-space exploration techniques discussed in this thesis could act as a base for future implementations of new algorithms.

COMMUNICATION PRIMITIVES

Protocol implementations that are generated by Prut4j have limited communication possibilities. At this time, we only support `send` and `receive` operations. The protocol implementations could be more powerful with support for more communication primitives like:

- **Sync.** Two threads send and receive a message at the same time.
- **Multiple receivers.** One thread can send a single message that will be read by multiple receiving threads.

DISTRIBUTED PROTOCOL IMPLEMENTATIONS

This research mainly aims at concurrent programs that execute with shared memory. This is a requirement for the strict protocol implementations (they share a state id). However, the liberal protocol implementations might be usable in distributed architectures. The protocol implementation queues could be replaced with queues that can communicate, for example, over a network.

A

SUPPLEMENTAL RESOURCES

Artifact	Location	Note
StateSpaceExploringThread	`\library\src\main\java\nl\florianslob\modelchecking\base\runtime\v2`	
NetworkTopologies .dcj files	`\code-generator\protocol_definitions\network_topologies`	
NetworkTopologies .owl files	`\tests\formulas\NetworkTopologies`	
Games .dcj files	`\code-generator\protocol_definitions\games`	
Game .owl files	`\tests\formulas`	Separate folder for every game
NPB .dcj files	`\code-generator\protocol_definitions\npb`	
NPB .owl files	`\tests\formulas\npb`	Separate folder for every game
NPB Result spreadsheets	`\benchmark-jar-files\results`	available in .xlsx and .csv format
Test execution summaries	`\tests\summaries`	

Table A.1: The supplemental materials contain a full copy of the Prut4j repository. This table outlines certain resources that have been mentioned in this report.

REFERENCES

[1] O. Lichtenstein A. and Pnueli. "Checking That Finite State Concurrent Programs Satisfy Their Linear Specification". In: *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '85. New Orleans, Louisiana, USA: ACM, 1985, pp. 97–107. ISBN: 0-89791-147-4. DOI: 10.1145/318593.318622.

[2] F. Arbab. "Reo: a channel-based coordination model for component composition". In: *Mathematical Structures in Computer Science* 14.3 (2004), pp. 329–366. DOI: 10.1017/S0960129504004153.

[3] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. "A Uniform Framework for Modeling and Verifying Components and Connectors". In: *Coordination Models and Languages*. Ed. by J. Field and V. T. Vasconcelos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 247–267. ISBN: 978-3-642-02053-7. DOI: 10.1007/978-3-642-02053-7_13.

[4] D. Castro, R. Hu, S. T. Q. Jongmans, N. Ng, and N. Yoshida. "Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 29:1–29:30. ISSN: 2475-1421. DOI: 10.1145/3290342.

[5] B. L. Chamberlain, D. Callahan, and H. P. Zima. "Parallel Programmability and the Chapel Language". In: *The International Journal of High Performance Computing Applications* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[6] S. Chandra, P. Godefroid, and C. Palm. "Software Model Checking in Practice: An Industrial Case Study". In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida: ACM, 2002, pp. 431–441. ISBN: 1-58113-472-X. DOI: 10.1145/581339.581393.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 244–263. ISSN: 0164-0925. DOI: 10.1145/5397.5399.

[8] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. "State space reduction using partial order techniques". In: *International Journal on Software Tools for Technology Transfer* 2.3 (1999), pp. 279–287. DOI: 10.1007/s100090050035.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. DOI: 10.1007/BFb0058022.

[10] K. E. Coons, S. Burckhardt, and M. Musuvathi. "GAMBIT: effective unit testing for concurrency libraries". In: *PPOPP*. ACM, 2010, pp. 15–24. DOI: 10.1145/1837853.1693458.

[11] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. "Memory Efficient Algorithms for the Verification of Temporal Properties". In: *CAV*. Vol. 531. Lecture Notes in Computer Science. Springer, 1990, pp. 233–242. DOI: 10.1007/BF00121128.

[12] E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Texts and Monographs in Computer Science. Springer, 1982. ISBN: 0-387-90652-5.

[13] E. W. Dijkstra. "The humble programmer". In: *Commun. ACM* 15.10 (1972), pp. 859–866. DOI: 10.1145/355604.361591.

[14] M. A. Frumkin, M. G. Schultz, H. Jin, and J. C. Yan. "Performance and Scalability of the NAS Parallel Benchmarks in Java". In: *IPDPS*. IEEE Computer Society, 2003, p. 139. DOI: 10.1109/IPDPS.2003.1213267.

[15] R. Gerth, D. A. Peled, M. Y. Vardi, and P. Wolper. "Simple on-the-fly automatic verification of linear temporal logic". In: *PSTV*. Vol. 38. IFIP Conference Proceedings. Chapman & Hall, 1995, pp. 3–18. ISBN: 978-0-387-34892-6. DOI: 10.1007/978-0-387-34892-6_1.

[16] P. Godefroid. "Software Model Checking: The VeriSoft Approach". In: *Formal Methods in System Design* 26.2 (Mar. 2005), pp. 77–101. ISSN: 1572-8102. DOI: 10.1007/s10703-005-1489-x.

[17] P. Godefroid and K. Sen. "Combining Model Checking and Testing". In: *Handbook of Model Checking*. Springer, 2018, pp. 613–649.

[18] A. Groce, D. Peled, and M. Yannakakis. "Adaptive Model Checking". In: *Logic Journal of the IGPL* 14.5 (Oct. 2006), pp. 729–744. ISSN: 1367-0751. DOI: 10.1093/jigpal/jzl007.

[19] R. Hamers and S. T. Q. Jongmans. "Discourje: Runtime Verification of Communication Protocols in Clojure". In: *TACAS (1)*. Vol. 12078. Lecture Notes in Computer Science. Springer, 2020, pp. 266–284. DOI: 10.1007/978-3-030-45190-5_15.

[20] G. J. Holzmann and W. S. Lieberman. *Design and validation of computer protocols*. Vol. 512. Prentice hall Englewood Cliffs, 1991.

[21] K. Honda, A. Mukhamedov, G. Brown, T. Chen, and N. Yoshida. "Scribbling Interactions with a Formal Foundation". In: *Distributed Computing and Internet Technology*. Ed. by R. Natarajan and A. Ojo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 55–75. DOI: 10.1007/978-3-642-19056-8_4.

[22] K. Honda, N. Yoshida, and M. Carbone. "Multiparty Asynchronous Session Types". In: *J. ACM* 63.1 (2016), 9:1–9:67. DOI: 10.1145/2827695.

[23] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. "Improved multithreaded unit testing". In: *SIGSOFT FSE*. ACM, 2011, pp. 223–233. DOI: 10.1145/2025113.2025145.

[24] R. Jhala and R. Majumdar. "Software Model Checking". In: *ACM Comput. Surv.* 41.4 (Oct. 2009), 21:1–21:54. ISSN: 0360-0300. DOI: 10.1145/1592434.1592438.

[25] S. T. Q. Jongmans. "Toward New Unit-Testing Techniques for Shared-Memory Concurrent Programs". In: *ICECCS*. IEEE, 2019, pp. 164–169. DOI: 10.1109/ICECCS.2019.00025.

[26] S. T. Q. Jongmans and F. Arbab. "PrDK: Protocol Programming with Automata". In: *TACAS*. Vol. 9636. LNCS. Springer, 2016, pp. 547–552. DOI: 10.1007/978-3-662-49674-9_33.

[27] S. Keshishzadeh, M. Izadi, and A. Movaghar. “A BÜChi Automata Based Model Checking Framework for Reo Connectors”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC ’12. Trento, Italy: ACM, 2012, pp. 1536–1543. ISBN: 978-1-4503-0857-1. DOI: 10.1145/2245276.2232021.

[28] S. Klüppelholz and C. Baier. “Symbolic model checking for channel-based component connectors”. In: *Science of Computer Programming* 74.9 (2009), pp. 688–701. DOI: 10.1016/j.entcs.2007.03.003.

[29] N. Kokash and F. Arbab. “Formal Design and Verification of Long-Running Transactions with Extensible Coordination Tools”. In: *IEEE Transactions on Services Computing* 6.2 (Apr. 2013), pp. 186–200. ISSN: 1939-1374. DOI: 10.1109/TSC.2011.46.

[30] N. Kokash, C. Krause, and E. de Vink. “Reo + mCRL2 : A framework for model-checking dataflow in service compositions”. In: *Formal Aspects of Computing* 24.2 (Mar. 2012), pp. 187–216. ISSN: 1433-299X. DOI: 10.1007/s00165-011-0191-6.

[31] J. Křetínský, T. Meggendorfer, and S. Sickert. “Owl: A Library for ω-Words, Automata, and LTL”. In: *ATVA*. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 543–550. DOI: 10.1007/978-3-030-01090-4_34.

[32] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. “Root causing flaky tests in a large-scale industrial setting”. In: *ISSTA*. ACM, 2019, pp. 101–111. DOI: 10.1145/3293882.3330570.

[33] H. Liang, L. Sun, M. Wang, and Y. Yang. “Deep Learning With Customized Abstract Syntax Tree for Bug Localization”. In: *IEEE Access* 7 (2019), pp. 116309–116320. DOI: 10.1109/ACCESS.2019.2936948.

[34] B. Long, D. Hoffman, and P. A. Strooper. “Tool Support for Testing Concurrent Java Components”. In: *IEEE Trans. Software Eng.* 29.6 (2003), pp. 555–566. DOI: 10.1109/TSE.2003.1205182.

[35] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. “An empirical analysis of flaky tests”. In: *SIGSOFT FSE*. ACM, 2014, pp. 643–653. DOI: 10.1145/2635868.2635920.

[36] Z. Merali. “Computational science: ...Error”. In: *Nature* 467 (2010), pp. 775–777. DOI: 10.1038/467775a.

[37] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. “Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code”. In: *ICSE*. 2012, pp. 727–737. DOI: 10.1109/ICSE.2012.6227145.

[38] D. L. Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules”. In: *Commun. ACM* 15.12 (1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623.

[39] A. Pnueli. “The Temporal Logic of Programs”. In: *FOCS*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[40] W. Pugh and N. Ayewah. “Unit testing concurrent software”. In: *ASE*. ACM, 2007, pp. 513–516. DOI: 10.1145/1321631.1321722.

[41] J. P. Queille and J. Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *International Symposium on Programming*. Ed. by M. Dezani-Ciancaglini and U. Montanari. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 337–351. ISBN: 978-3-540-39184-5. DOI: 10.1007/3-540-11494-7_22.

[42] D. Silva, L. Teixeira, and M. d’Amorim. “Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker”. In: *ICSME*. IEEE, 2020, pp. 301–311. DOI: 10.1109/ICSME46990.2020.00037.

[43] F. J. Slob and S. T. Q. Jongmans. “Prut4j: Protocol Unit Testing fo(u)r Java”. In: Article has been accepted for the 14th IEEE Conference on Software Testing, Verification and Validation (ICST). 2021. URL: https://sungshik.github.io/papers/icst2021.pdf.

[44] S. Steenbuck and G. Fraser. “Generating Unit Tests for Concurrent Classes”. In: *ICST*. IEEE Computer Society, 2013, pp. 144–153. DOI: 10.1109/ICST.2013.33.

[45] T. Tu, X Liu, L. Song, and Y. Zhang. “Understanding Real-World Concurrency Bugs in Go”. In: *ASPLOS*. ACM, 2019, pp. 865–878. DOI: 10.1145/3297858.3304069.

[46] B. van Veen and S. T. Q. Jongmans. “Modular Programming of Synchronization and Communication Among Tasks in Parallel Programs”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018, pp. 425–435. DOI: 10.1109/IPDPSW.2018.00077.

[47] M. Y. Vardi and P. Wolper. “An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)”. In: *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society, 1986, pp. 332–344. URL: https://orbi.uliege.be/handle/2268/116609.

[48] N. Yoshida, R. Hu, R. Neykova, and N. Ng. “The Scribble Protocol Language”. In: *Trustworthy Global Computing*. Ed. by M. Abadi and A. L. Lafuente. Cham: Springer International Publishing, 2014, pp. 22–41. ISBN: 978-3-319-05119-2. DOI: 10.1007/978-3-319-05119-2_3.